

OAuth 2.0

Proteja suas aplicações com o
Spring Security OAuth2



Sumário

- [ISBN](#)
- [Agradecimentos](#)
- [Sobre o autor](#)
- [Sobre o livro](#)
- [1. Introdução](#)
- [2. O framework de autorização OAuth 2.0](#)
- [3. Iniciando o projeto de exemplo](#)
- [4. Proteja a aplicação com Spring Security](#)
- [5. Exemplo de aplicação cliente](#)
- [6. O Spring Security OAuth2](#)
- [7. Password credentials grant type](#)
- [8. Funcionamento interno do Spring Security OAuth2](#)
- [9. Authorization Code grant type](#)
- [10. Implicit grant type para aplicações no Browser](#)
- [11. Client Credentials](#)
- [12. Refresh Tokens e escopos para aumentar a segurança](#)
- [13. Adicionando suporte a banco de dados no projeto](#)
- [14. Client com o Spring Security OAuth2](#)
- [15. Alternativas para validação de tokens](#)
- [16. OAuth 2.0 como base para autenticação](#)
- [17. Algumas considerações sobre segurança](#)
- [18. Bibliografia](#)

ISBN

Impresso e PDF: 978-85-94188-12-0

EPUB: 978-85-94188-13-7

MOBI: 978-85-94188-14-4

Você pode discutir sobre este livro no Fórum da Casa do Código:
<http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

Agradecimentos

Em primeiro lugar, agradeço aos meus pais pela dedicação empregada em minha educação, pelo apoio e incentivo a tudo que eu quis estudar e por serem, para mim, grande fonte de inspiração. Não posso deixar de citar também a paciência de minha companheira Janine, por sempre me apoiar em meus trabalhos e estudos. Agradeço também ao Jorge Acetozi, por me incentivar na escrita do livro, e ao Rafael Chinelato, por ajudar na revisão técnica de alguns capítulos.

Agradeço à Casa do Código por tornar possível a publicação deste material e por permitir que tantos livros sejam publicados sem burocracia, facilitando o compartilhamento de conhecimento. Agradeço ao Adriano Almeida e à Vivian Matsui pela oportunidade e apoio durante a escrita do livro.

Também agradeço ao leitor que se interessou pelo assunto ao qual tanto me dediquei, com o intuito de facilitar o entendimento consolidando o que aprendi na prática e estudando especificações, livros sobre segurança, Spring e OAuth.

Sobre o autor

Eu sou o Adolfo Eloy, graduado em Ciência da Computação em 2007, pela FAC-FITO, e pós-graduado em Soluções de Sistemas Corporativos em Java em 2010, pela FIAP. Comecei a trabalhar com desenvolvimento de software profissionalmente em 1999, criando soluções que executavam diretamente na máquina do usuário com mecanismos muito simples de autenticação. Comecei a participar do desenvolvimento de sistemas web lá em 2003, quando ainda não se ouvia falar de OAuth.

Já trabalhei no desenvolvimento de sistemas para empresas de médio porte, e-commerce, meios de pagamento e soluções corporativas para instituições financeiras. Em cada projeto, tive a oportunidade de conhecer e ajudar a definir diferentes arquiteturas, sempre atento aos erros e acertos.

Acredito que o conhecimento tem muito valor quando aplicado e divulgado, seja através do ensino ou da publicação de conteúdo através de blog posts, palestras, artigos para revistas e livros.

Comecei escrevendo posts para o meu primeiro blog, em <http://aeloy.blogspot.com>, que já não é mais atualizado dando lugar ao mais recente <http://adolfoeloy.com.br>.

Além de posts nos blogs, também escrevo artigos para a revista JavaMagazine, e gosto de contribuir sempre que posso em fóruns de discussão sobre desenvolvimento de software.

Sobre o livro

Quando tive a necessidade de proteger recursos de uma aplicação com OAuth 2.0, em um projeto em que trabalhei, ele já seguia a stack do Spring, sendo natural utilizar a implementação de OAuth 2.0 do próprio **Spring Security OAuth**. Porém, ler a documentação da biblioteca e escrever código não foram suficientes para que eu tivesse um entendimento satisfatório sobre o assunto. Por isso, acabei buscando referências em livros sobre segurança e nas especificações descritas através de RFCs, em especial a RFC 6749.

RFC, ou *Request For Comments*, trata-se de documentos que podem ser criados por comunidades ou empresas para especificar protocolos, algoritmos, frameworks e tudo mais que possa ser padronizado.

A motivação para a escrita do livro foi condensar, em um material, teoria e prática a respeito de OAuth 2.0, utilizando o projeto Spring Security OAuth2. Ler especificações, por exemplo, pode ser muito cansativo para algumas pessoas (e ainda não há códigos de exemplo para facilitar o entendimento). Em contrapartida, contar apenas com exemplos de código da internet também pode não ser o melhor caminho (ainda mais quando se trata de segurança).

Ao ler este livro, o desenvolvedor poderá entender como usar o OAuth 2.0 em cenários específicos, bem como entender o porquê de proteger recursos com OAuth 2.0. Durante o dia a dia e mesmo lendo sobre o OAuth na internet, percebe-se que, apesar de o assunto não ser novo, ainda existe bastante confusão e até mesmo cenários de mau uso da tecnologia. A própria especificação do OAuth 2.0 já nasceu de uma forma um pouco confusa e até contraditória para alguns ex-contribuidores da RFC 6749.

Como exemplo de confusão, muitos ainda podem se perguntar se OAuth 2.0 é um protocolo ou um framework. Além disso, alguns

desenvolvedores também acreditam que OAuth deve ser utilizado para autenticação. Será que deve ser usado para autenticação ou autorização? O papel do livro é desmistificar todas essas perguntas, como alertar para o uso seguro da tecnologia.

Ainda em 2017, muitas empresas que utilizam OAuth 2.0 deixam brechas de segurança justamente por não se atentarem a detalhes tão pequenos, mas de extrema importância.

Público-alvo

O livro é indicado tanto para desenvolvedores que ainda não conhecem OAuth 2.0 quanto para aqueles que já vêm utilizando pontualmente em algumas aplicações. Acredito que mesmo desenvolvedores que já o tenham usado em seus projetos podem se beneficiar da leitura, pelo fato de serem apresentados vários fluxos diferentes de utilização do protocolo e pontos importantes de segurança.

Apesar de o protocolo OAuth 2.0 estar presente nos projetos há bastante tempo, volta e meia ainda aparecem notícias em blogs de segurança apresentando brechas em sistemas grandes que protegem suas APIs com OAuth.

Como os exemplos são dados utilizando uma aplicação construída em Java com o framework Spring, é interessante que você tenha conhecimento básico do funcionamento do Spring. Não é necessário que você tenha conhecimento de todo o framework Spring, mas é importante conhecer como funciona o mecanismo de injeção de dependências oferecido pelo framework, e como o Spring gerencia os objetos mantidos no contexto da aplicação (objetos que também são chamados de *beans* dentro do framework).

Quanto ao Spring MVC e ao Spring Security, um conhecimento básico também ajuda. Caso não tiver experiência com esses frameworks, não se preocupe, pois sempre que possível farei uma introdução breve sobre o assunto que estiver sendo abordado.

Também é interessante que você já tenha uma certa familiaridade com JPA, já que os exemplos apresentados usam um banco de dados relacional. Para ajudá-lo durante a leitura, deixei todos os exemplos utilizados no livro disponíveis no GitHub.

Alguns detalhes pontuais sobre Spring Security são apresentados no decorrer do livro, porém o funcionamento do Spring e do Spring MVC não são abordados, pois para isso seria necessário escrever um outro livro. Aliás, quanto ao Spring MVC, a Casa do Código já possui uma publicação bem legal do Alberto Souza que pode ser conferida em <https://www.casadocodigo.com.br/products/livro-spring-mvc>.

Código-fonte

Para facilitar o entendimento sobre o assunto, disponibilizei os exemplos de código em um projeto no GitHub. Eles podem ser baixados em <https://github.com/adolfoweloy/livro-spring-oauth2>.

A ideia é que você utilize esses projetos durante a leitura do livro, pois neles mostro como proteger uma API com o protocolo OAuth 2.0. Dentro do diretório principal `livro-spring-oauth2`, estão disponíveis os exemplos usados no livro, agrupados em subdiretórios e organizados por capítulo.

CAPÍTULO 1

Introdução

Quando comecei a trabalhar com desenvolvimento de software, os primeiros projetos de que participei se tratavam de aplicações que rodavam na máquina do usuário. Neles, uma rede era configurada apenas para permitir que o sistema pudesse acessar o banco de dados. Quase tudo que o sistema precisava para funcionar estava ali na máquina do próprio usuário.

Em cenários mais complexos, nessa época participei de projetos nos quais a regra de negócio da aplicação era distribuída em serviços que podiam ser acessados remotamente utilizando arquiteturas proprietárias como COM, CORBA e RMI. Estavam todos falando de componentização e aplicações distribuídas. Com os módulos distribuídos em um servidor de componentes, eles eram reutilizados facilitando a construção de sistemas robustos.

Era uma época interessante pois, ao mesmo tempo em que muitos desenvolvedores estavam empolgados com componentização e sistemas distribuídos, aplicações para a web também eram construídas e muitos já estavam considerando a integração entre sistemas diferentes através de protocolos baseados em HTTP. SOAP é um grande exemplo de protocolo que chamou muito a atenção das empresas em meados de 2000.

De lá para cá, as formas de se integrar sistemas evoluíram bastante com muitas aplicações adotando a arquitetura REST (aqui não entro no mérito sobre o nível de maturidade da arquitetura REST). Mas por que existe tanto interesse em integração entre sistemas de diferentes empresas?

Para se ter uma ideia, ao construir um sistema de comércio eletrônico, geralmente é necessário interagir com sistemas de pagamento, serviços de análise de fraude, armazenamento, busca,

além de poder fornecer os próprios serviços através de APIs (*Application Programming Interface*). Imagine tentar fazer tudo isso em um único sistema. Seria impossível lançar um produto a tempo de atender às necessidades do momento.

O interesse em reutilização de serviços é tão grande que existem empresas que têm seu lucro baseado nas APIs que disponibilizam. É muito interessante observar como o número de APIs aumenta dia a dia analisando alguns números no site ProgrammableWeb (<http://www.programmableweb.com/api-research>). O site fornece uma plataforma para consulta e registro de APIs públicas, e o legal é que nele também podemos obter alguns dados estatísticos, como mostra a figura a seguir.

Quantidade de APIs registradas a cada três anos

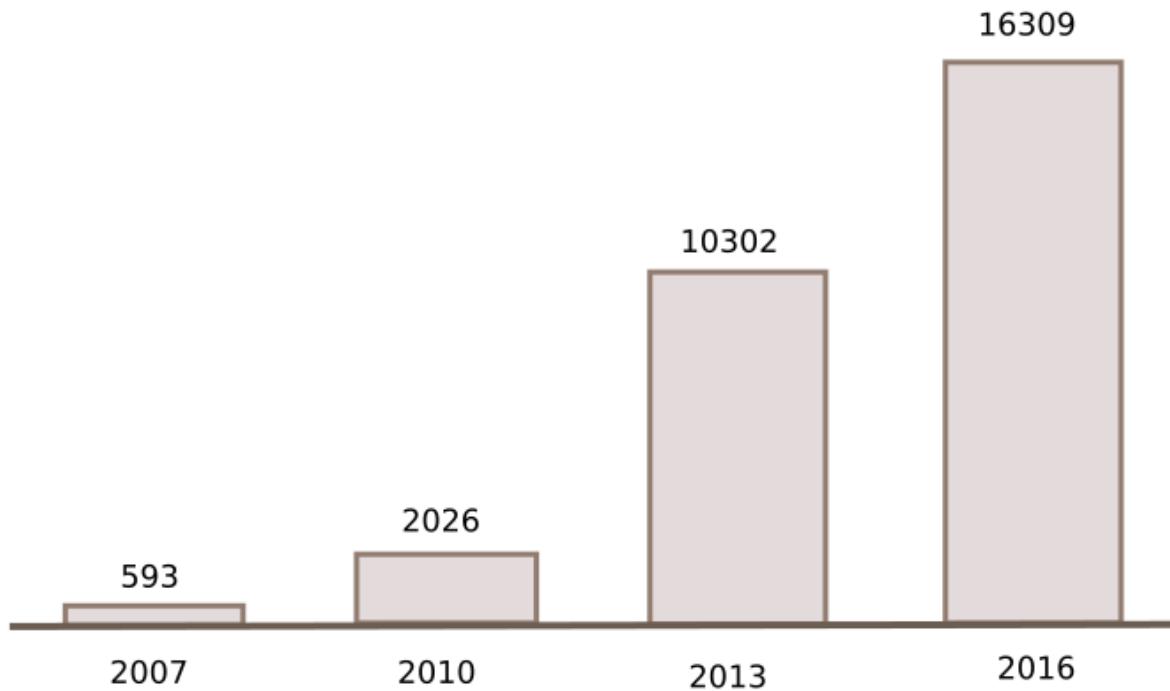


Figura 1.1: Quantidade de APIs registradas no site ProgrammableWeb

1.1 Como as APIs são protegidas

Quando contei um pouco sobre minha experiência em projetos mais antigos rodando em desktop, como as aplicações eram protegidas de usuários mal-intencionados? Para acessar o sistema, cada usuário tinha uma conta criada pelo próprio departamento de tecnologia, ou através de uma tela de administração na qual um usuário com mais permissões poderia criar as contas dos outros usuários.

De certa forma era bem simples realizar a **autenticação** de um usuário, ou seja, era simples identificar se o usuário que estava acessando o sistema era mesmo quem dizia ser através de um login e uma senha. Talvez o maior nível de dificuldade estivesse em definir as políticas de segurança que dizem o que o usuário pode fazer no sistema, ou em outras palavras, políticas de **autorização**.

Em sistemas *desktop*, apesar de existir uma preocupação com segurança, quando um usuário acessava o sistema, suas credenciais não trafegavam pela internet assim como nos sistemas atuais integrados através de tantas APIs. Com o uso das APIs, não somente usuários finais acessam o sistema, mas também outras aplicações. Temos muitos exemplos de aplicações web que usam serviços de pagamento, e aplicações mobile que acessam bases de produtos em sistemas de comércio eletrônico e IoT (*Internet Of Things*).

Inicialmente, para proteger uma API, podemos pensar em definir um usuário e senha para outra aplicação e realizar autenticação e autorização, assim como se faz durante um login normal de usuário. Podemos também pensar em entregar uma chave secreta, como um *token*, para a aplicação que acessa nossas APIs. Porém, além de autenticação e autorização, é preciso considerar alguns princípios primordiais de segurança de sistemas conhecidos como **confidencialidade, integridade e disponibilidade**.

Dados confidenciais como senhas, chaves de cartão de crédito e dados bancários, por exemplo, precisam ser protegidos para que não possam ser lidos por usuários ou sistemas não autorizados, seguindo para isso o princípio da **confidencialidade**. Para garantir a confidencialidade dos dados, podemos utilizar **criptografia** para protegê-los quando persistidos ou trafegados pela rede (usando SSL ou TLS).

Outro princípio importante é o da **integridade**, que visa garantir que os dados não sofram alterações não autorizadas. Uma forma de validar se uma mensagem foi alterada é através de assinatura. Para gerar uma assinatura, é necessário contar com uma função que gera um *hash* utilizando um cálculo que combina uma chave e a mensagem a ser assinada.

Ao receber a mensagem, um servidor pode fazer a validação gerando um *hash* com a mesma chave usada durante o processo de assinatura, e finalmente comparar se o *hash* gerado é igual ao recebido na assinatura da mensagem. O protocolo TLS faz algo parecido para garantir a integridade da mensagem durante o tráfego pela rede.

FUNÇÃO HASH

Basicamente, um hash se trata de uma cadeia de caracteres gerada a partir de outra cadeia de caracteres. Um hash é gerado a partir de um algoritmo executado por uma determinada função, de modo que o resultado dessa função geralmente é uma string com tamanho fixo, podendo ser menor do que a string de entrada. Caso queira saber mais sobre o assunto, acesse <https://goo.gl/WjDo3f>.

Também é necessário se preocupar com a **disponibilidade** do sistema. Aplicações que podem ser acessadas através da internet são alvos de ataques que visam deixar o sistema inoperante, e se proteger disso também é um grande desafio. Para isso, muitas

estratégias podem ser adotadas, como por exemplo, utilizar *rate limit*, uma forma de limitar a quantidade de requisições que uma aplicação pode receber por um determinado intervalo de tempo.

Entretanto, para manter um sistema disponível, também precisamos levar em consideração outras questões, como redundância e atualização do sistema operacional, para evitar quedas desnecessárias no sistema.

A ideia aqui não é entrar nos detalhes sobre segurança de sistemas pois, para isso, seria necessário escrever outro livro. O importante é que o leitor tenha uma visão geral sobre os cenários complexos que temos hoje em dia quando estamos desenvolvendo uma aplicação para a web. A larga utilização de APIs para integração entre sistemas trouxe mudanças significativas na forma como protegemos nossas aplicações.

1.2 Acessando uma API de um jeito diferente

Voltando para os conceitos de *autenticação* e *autorização*, geralmente os sistemas validam o acesso de usuários finais (humanos) e o acesso de aplicações. Pessoas ou sistemas podem manipular os próprios dados, seja através da interação com uma interface gráfica ou de forma programática através de uma API. Porém, uma outra forma de interação ganhou força com o advento das redes sociais.

É comum hoje em dia sistemas que acessam APIs de redes sociais para obter alguns dados de seus usuários (tudo isso autorizado pelo próprio usuário da rede social). Além de consultar dados, os sistemas também podem realizar operações em benefício do usuário de uma rede social, como publicar algum conteúdo na linha do tempo, por exemplo. Esse cenário traz outros desafios quando o assunto é segurança, conforme veremos a seguir.

1.3 Repassando credenciais, um exemplo de anti-pattern

Pensando no cenário apresentado anteriormente, no qual um sistema acessa os dados de um usuário em outro sistema, vamos apresentar uma aplicação fictícia batizada com o nome **bookserver**, que será a base dos exemplos usados no livro. A aplicação de exemplo se trata de um sistema web que permite aos usuários cadastrar e consultar livros que já leram.

Até aqui nada fora do normal, pois trata-se de uma aplicação bem simples com alguns cadastros básicos e uma tela de autenticação em que o usuário coloca seu login e senha para acessar o sistema. Agora imagine que o mesmo usuário também possui uma conta em uma rede social que deseja integrar com nosso sistema de livros.

No cenário que estamos imaginando, a tal rede social deseja consultar os livros que o usuário já leu para poder exibi-los como conteúdo na linha do tempo do usuário, conforme mostrado na figura a seguir:

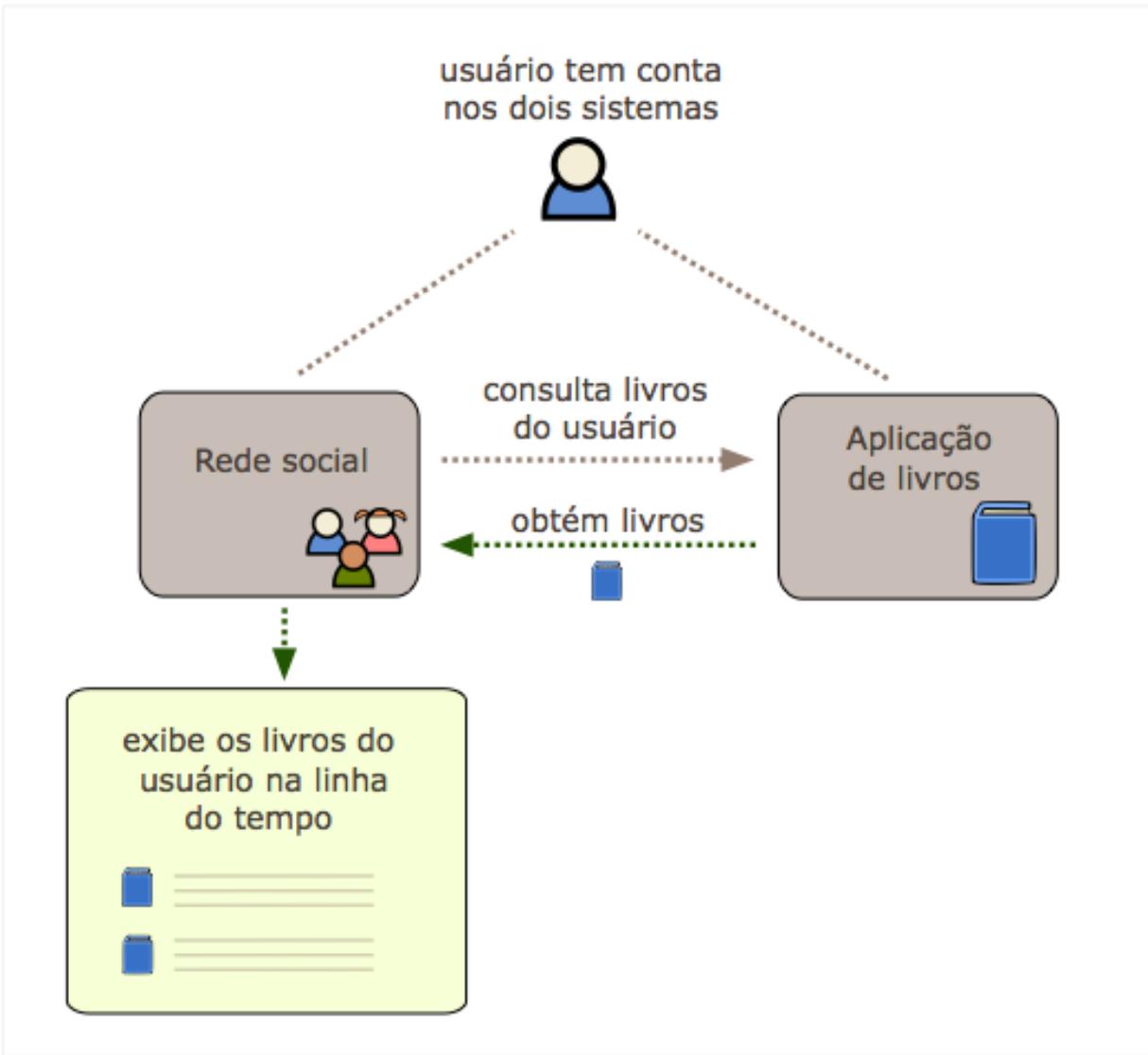


Figura 1.2: Exemplo de integração entre rede social e a aplicação de exemplo

Agora precisamos pensar em como a aplicação vai acessar os livros do usuário. Uma solução bem simples aqui seria utilizar o mesmo login e senha que o usuário possui na rede social para acessar a aplicação de livros. Porém, não podemos considerar que o usuário manterá login e senha iguais em todas aplicações. Se o usuário não possui login e senha iguais nos dois sistemas, então a rede social precisa repassar as credenciais que o usuário tem na aplicação de livros.



Figura 1.3: Repassando as credenciais do usuário

Porém, como a aplicação de livros saberá se quem está acessando a aplicação é o usuário ou a rede social? Ao repassar as credenciais, a rede social teria os mesmos acessos que o usuário final, podendo então cadastrar novos livros ou até mesmo excluir registros em nome do usuário, o que não é nada desejável.

Precisamos de uma solução na qual o usuário não precise repassar suas credenciais. Um tipo de credencial temporária poderia ser gerado para a aplicação que acessa os dados em nome do usuário. Ao gerar essa credencial temporária, ele poderia escolher as permissões que deseja dar para o outro sistema, e depois repassar essa credencial para a aplicação que vai acessar seus dados.

1.4 Delegação de acesso a recursos protegidos

Através de um exemplo encontrado na vida real, vamos enxergar melhor a solução técnica que estamos buscando. Vamos fazer uma analogia com condomínios, em que um morador permite a entrada de prestadores de serviço em seu apartamento para realização de manutenções. Em alguns condomínios, é comum o morador solicitar na administração do prédio um cartão magnético de visitantes que ele pode entregar para um prestador de serviços poder passar pela

portaria durante um período determinado e com permissões específicas.

Como exemplo de permissões, podemos considerar que o prestador de serviços terá acesso apenas a entrar no apartamento, mas não tem acesso a utilizar a quadra nem a piscina do prédio. Nessa analogia, o morador (ou usuário da aplicação de livros) não entrega seu próprio cartão para o prestador de serviços. Em vez disso, ele vai até a administração onde é identificado como morador e **autoriza** a entrada de um prestador de serviços específico.

A administração entrega então um cartão que representa a **autorização** de entrada do prestador para o morador que, por sua vez, entrega ao prestador de serviços. É importante ter em mente que se outra pessoa pegar esse cartão, ela pode entrar no prédio passando-se pelo prestador de serviços autorizado inicialmente.

Voltando para o sistema de livros, o componente faltante no esquema de identificação que estamos procurando pode ser algo parecido com o cartão de visitantes. Mas como representar esse cartão em nossos sistemas? Podemos fazê-lo através de um **token de acesso**.

Um token de acesso nada mais é do que uma chave representada por uma cadeia de caracteres ou números que pode ser gerada pela aplicação de livros e entregue para a rede social. A figura a seguir ilustra como a rede social poderia interagir com a aplicação de livros utilizando um token de acesso fornecido pelo usuário.

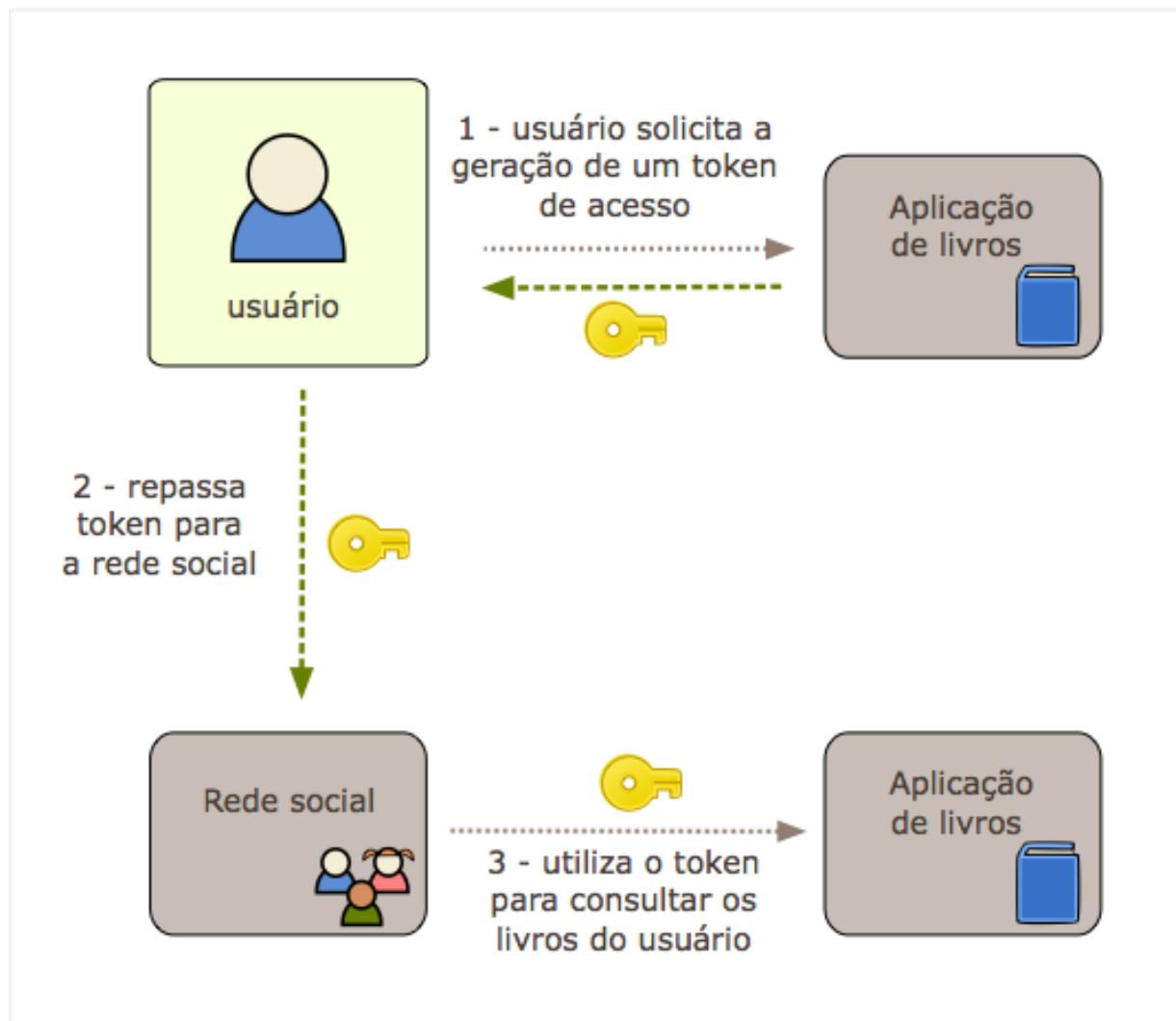


Figura 1.4: Token de acesso fornecido pelo usuário

Ao entregar o token de acesso para a rede social, o usuário está **delegando o acesso** aos seus recursos para um terceiro poder realizar consultas em seu nome.

Gerenciamento do token de acesso

A questão aqui é: como esse token de acesso é entregue à rede social? Conforme mostra a figura anterior, o próprio usuário entrega esse token. Para isso, a rede social poderia ter uma interface na qual ele informaria manualmente o token de acesso. Mas será que

isso seria bom para o usuário? Essa solução traz alguns problemas de usabilidade.

Imagine o usuário ter de gerar tokens de acesso para todo tipo de sistema que realizar integração com o sistema de livros. Seria muito fácil ele acabar informando tokens de acesso inválidos entre aplicações. Caso esse token de acesso algum dia seja alterado, o usuário precisará informar o novo para a aplicação que se integra com o sistema de livros. Deixar essa responsabilidade com ele não é o tipo de solução que queremos para a integração com o sistema de livros.

A responsabilidade de gerenciar o token de acesso pode ser feita de forma transparente para o usuário caso o próprio sistema de livros possa gerar o token de acesso, e então enviar para a aplicação que deseja realizar a integração.

1.5 Soluções para delegação de acesso

Já vimos que repassar as credenciais do usuário para outra aplicação não é uma boa solução. Caso você fosse criar uma conta no Twitter em meados de 2006, notaria que o Twitter oferecia durante o cadastro uma opção para informar as credenciais de sua conta de e-mail. Dessa forma, ele poderia enviar convites de cadastro para os seus contatos. Apesar de o usuário ser informado que o e-mail e a senha não seriam mantidos no banco de dados do Twitter, o usuário precisaria confiar que seus dados não seriam divulgados e nem usados de maneira inapropriada. Essa solução logo foi substituída utilizando o conceito de delegação de acesso.

Nessa mesma época, muitas empresas criaram suas próprias soluções para delegação de acesso. O Google, por exemplo, para permitir que outras aplicações criassem eventos no Google Calendar também precisava de uma solução para evitar que os

usuários repassassem suas credenciais, levando à criação de dois protocolos de segurança conhecidos como *ClientLogin* e *AuthSub*.

O Yahoo! também tinha seu próprio mecanismo chamado de *BBAuth*. Com tantas soluções diferentes, a falta de padrão foi reconhecida por membros de empresas como o Twitter que, em conjunto com uma pequena comunidade de desenvolvedores web, criaram o protocolo OAuth 1.0 para padronizar uma solução para delegação de acesso.

1.6 OAuth como solução e sua evolução para o OAuth 2.0

O OAuth é um protocolo que nasceu basicamente para resolver o problema de ter de repassar as credenciais do usuário para outra aplicação, ou seja, para facilitar o desenvolvimento de aplicações que acessam dados em outras aplicações, em benefício de um usuário em comum utilizando delegação de acesso. O protocolo OAuth padroniza a forma de se obter um token de acesso de forma transparente para o usuário, sem que este precise gerenciar os tokens de acesso.

Devido ao grande número de integração entre sistemas através de APIs, o protocolo OAuth tornou-se largamente utilizado (principalmente para aplicações que interagiam com serviços do Google, Twitter ou Facebook). Porém, a especificação do protocolo OAuth 1.0 — que pode ser lida em <https://tools.ietf.org/html/rfc5849> — já não é mais suportada na maioria dos projetos, devido à evolução do protocolo para sua versão 2.0.

Antes de chegar ao OAuth 2.0 (cuja especificação pode ser lida em <https://tools.ietf.org/html/rfc6749>), o protocolo passou por diversas mudanças devido a algumas limitações identificadas no OAuth 1.0. As principais mudanças que serão abordadas adiante são:

- Suporte a diferentes tipos de aplicação;
- Facilidade para o desenvolvedor;
- Extensibilidade.

1.7 Suporte a diferentes tipos de aplicação

A especificação do protocolo OAuth 1.0 permite a utilização de delegação de acesso para aplicações web padrão, ou seja, aplicações web que permitem que o usuário seja redirecionado de uma aplicação para a outra, para possibilitar a troca de tokens de acesso de forma transparente. O OAuth 1.0 é uma boa solução para delegação de acesso considerando o fluxo básico para uma aplicação web, porém existem outros cenários que também precisam ser tratados.

O OAuth 1.0 não define uma forma de se obter e usar tokens de acesso quando se fala em aplicativos nativos para smartphone e aplicações JavaScript que executam direto no navegador. Além disso, ele também não cobre cenários em que uma aplicação precisa acessar dados em benefício próprio através de uma API. Diante disso, ficou evidente a necessidade de suporte a diferentes tipos de aplicação, sendo esse um fator relevante para o surgimento do OAuth 2.0.

Diferença entre clientes confidenciais e públicos

A necessidade de se tratar diferentes fluxos para obtenção e utilização de um token de acesso vem tanto de limitações técnicas quanto de segurança. Uma aplicação nativa, por exemplo, não tem o suporte de um browser para permitir navegação para o servidor OAuth e nem suporte ao redirecionamento.

Algumas aplicações *mobile* recorrem a web views para poder usar um fluxo básico parecido com o descrito na especificação do OAuth

1.0. Esse tipo de solução resolve o problema em partes, mas vai de encontro com a experiência do usuário. Já uma aplicação JavaScript que executa diretamente no browser conta com o suporte dele para navegação entre diferentes sistemas, porém vários problemas de segurança podem aparecer caso não seja utilizado um fluxo apropriado para obtenção de um token de acesso.

No que tange a segurança da aplicação, é crucial que os fluxos do protocolo de delegação considerem a capacidade que cada aplicação cliente tem de proteger credenciais ou tokens de acesso. Cada tipo de cliente, ou **client type**, pode ser classificado como **confidencial** ou **público** conforme a especificação do OAuth 2.0. Como exemplo de cliente confidencial, podemos citar aplicações web que realizam processamento do lado do servidor. Esse tipo de aplicação tem a capacidade de manter as credenciais e o token de acesso protegidos no servidor.

Agora vamos pensar no exemplo de aplicação JavaScript que executa diretamente no navegador do usuário. Em uma aplicação desse tipo, não temos um servidor para armazenar os dados confidenciais, ou seja, estes ficam disponíveis no código-fonte da página sendo acessada no *browser*. Desse modo, como os dados estão presentes no browser, eles podem ser acessados publicamente, seja pelo usuário ou por algum aplicativo que possa ler o conteúdo do browser.

Aplicativos móveis, ou até mesmo sistemas nativos executando na máquina do usuário, também não são capazes de manter credenciais de maneira segura, pois um sistema instalado, seja no computador ou no smartphone, pode ser descompilado revelando assim as credenciais da aplicação. Apesar de existirem alternativas para guardar credenciais, como guardar em um servidor na nuvem, ainda assim esses tipos de *Clients* são considerados como **públicos**.

Categorizando os tipos de clientes

Uma vez que os *client types* estão classificados como *público* ou *confidenciais*, é preciso definir a forma de se obter e utilizar tokens de acesso para cada tipo de aplicação. O OAuth 2.0 define fluxos específicos para cada tipo de aplicação considerando para isso o perfil de cada *Client*.

A especificação OAuth 2.0 usa o termo **client profile** para categorizar as aplicações de acordo com o modo como cada uma funciona. O OAuth 2.0 define três **client profiles** conforme mostrado a seguir:

- **Aplicações web:** consideradas como tipo de cliente *confidential*, são aplicações web que seguem o padrão de arquitetura básico com um servidor processando os dados e a navegação do usuário. Como executam o processamento no servidor, são aplicações que possuem a capacidade de manter a confidencialidade de credenciais e tokens de acesso.
- **Aplicações baseadas em browser:** como exemplo desse tipo de aplicação, podemos imaginar aplicações que executam diretamente no browser. Devido à natureza desse tipo de aplicação, esta não possui a capacidade de esconder do usuário as credenciais e tokens de acesso. Por isso, são consideradas como clientes *public* com relação ao nível de confidencialidade que possuem.
- **Aplicações nativas:** também são clientes considerados como *public*, porém ainda possuem a capacidade de manter um certo nível de proteção com relação as credenciais e tokens de acesso. Apesar de "esconder" dados confidenciais do usuário final, um usuário com um pouco mais de conhecimento pode descompilar o projeto para obter os dados que precisa para realizar fraudes no sistema.

Mesmo com tal fragilidade, diferentemente de aplicações que executam diretamente no browser, uma aplicação nativa pode executar em um processo protegido dependendo do sistema

operacional. Desse modo, outras aplicações que executam no mesmo dispositivo não conseguem acessar os dados de outras aplicações que executam em processos diferentes.

1.8 Facilidade para o desenvolvedor

Outro fator importante que foi visto como uma limitação do protocolo OAuth 1.0 diz respeito ao fato de ser necessário utilizar assinaturas no fluxo de obtenção de tokens de acesso. Tal fator foi visto como limitação porque, para gerar uma assinatura, é preciso implementar uma sequência de operações não muito triviais.

Não entraremos em detalhes sobre como implementar a geração de uma assinatura, pois o foco deste livro é o OAuth 2.0 que dispensa o uso dela. Mas de maneira bem abstrata, para se gerar uma assinatura é preciso usar uma *string base* formada pela concatenação de campos da requisição com o método HTTP e a URL da requisição codificada com *percent encoding*. Uma vez montada a string base, é preciso assiná-la utilizando `HMAC-SHA1` ou `RSA-SHA1`.

O OAuth 1.0 também define um método mais fácil de assinatura que é o `PLAINTEXT`. Esse método apenas concatena a chave secreta da aplicação cliente com um `&`.

Sem entrar no mérito do que é fácil ou difícil, o fato é que, uma vez implementado o algoritmo para geração da assinatura, este poderia ser reutilizado em outras aplicações. Ou seja, o desenvolvedor da aplicação cliente deveria ter esse trabalho uma única vez, e depois empacotar a solução como uma biblioteca para uso posterior. Porém, com o objetivo de facilitar a vida do desenvolvedor da aplicação *Client*, o protocolo OAuth 2.0 dispensa o uso de assinatura. Para isso, ele obriga que todas as requisições sejam

feitas utilizando uma camada de transporte segura através de TLS ou SSL.

TLS E SSL

TLS (ou *Transport Layer Security*) é desenvolvido em cima do protocolo SSL, ou *Secure Socket Layer*. O TLS, assim como o SSL, são protocolos usados para proteger requisições realizadas através da internet. O uso mais comum é para proteger a comunicação através do protocolo HTTPS.

O protocolo HTTPS usa TLS ou SSL para proteger as requisições. A principal diferença entre o TLS e o SSL é que o TLS permite autenticação mútua, de modo que não somente o servidor precise provar sua identidade, mas também o client.

Remover a assinatura do protocolo não foi uma característica muito bem recebida pelo Eran Hammer (autor da especificação do OAuth 1.0 e líder da especificação do OAuth 2.0). No post *OAuth 2.0 and the Road to Hell*, Eran Hammer (2012) questiona o fato de dispensar o uso da assinatura como sendo um problema de segurança. Além disso, apesar da facilidade de o desenvolvedor do *Client* não precisar mais gerar assinaturas, Hammer cita que, devido à necessidade de expirar tokens de acesso com o OAuth 2.0, o desenvolvedor passa a ter o trabalho de gerenciar **refresh tokens** para poder obter novos tokens de acesso (mais sobre refresh token será abordado mais adiante).

É importante lembrar de que a especificação do OAuth 2.0 traz alguns pontos de extensão que permite a customização e a construção de novos protocolos que tenham o OAuth 2.0 como base. Portanto, caso a falta de uma assinatura seja realmente um problema, é possível usar outros protocolos em conjunto com o OAuth 2.0.

1.9 Extensibilidade

Se o leitor tiver o interesse em dar uma lida na RFC 6749 (especificação do OAuth 2.0), poderá notar que, no título da especificação, o OAuth 2.0 é descrito como um **framework** de autorização. Diante desse título, pode então surgir a seguinte dúvida: OAuth 2.0 é um protocolo ou um framework?

Ao responder essa pergunta, podemos entender o real propósito da extensibilidade oferecida pelo OAuth 2.0. Mas antes de encontrarmos a resposta, quero deixar bem clara a diferença entre protocolo e framework de maneira bem direta.

- **Protocolo:** conjunto de normas e regras;
- **Framework:** uma solução genérica que serve como guia para construção de algo (no nosso caso, a construção de uma solução para delegação de acesso a recursos protegidos).

Diante da extensibilidade oferecida pelo OAuth 2.0 e por possuir alguns componentes opcionais, é muito justo que o título cite o OAuth 2.0 como um **framework** de autorização, e não um protocolo. Portanto, apesar de usarmos o termo protocolo, tenha em mente que o OAuth 2.0 é um framework. Mais adiante, nos próximos capítulos, veremos que existem vários aspectos que estão fora do escopo da especificação do OAuth 2.0.

Devido a utilização de componentes opcionais e a questão da extensibilidade, diferentes implementações do OAuth 2.0 podem não ser compatíveis. Como exemplo, caso um desenvolvedor crie uma aplicação que faça integração com um servidor protegido por OAuth 2.0 e, no futuro, houver a necessidade de interagir com outro servidor protegido por OAuth 2.0, é possível que o código escrito para obter e usar tokens de acesso pare de funcionar e tenha de ser adaptado.

Endpoints, a forma como o client se registra no servidor, a forma como o client se autentica para obter um token de acesso e outros

fatores podem ser totalmente diferentes, dependendo de como o servidor protegido por OAuth 2.0 foi implementado. A questão da falta de interoperabilidade pode ser vista de maneira bastante negativa, conforme mostrado pelo próprio Eran Hammer em um vídeo disponível em <https://vimeo.com/52882780>.

O que é visto por muitos como algo negativo também pode ser visto como positivo por outros. A própria especificação do OAuth 2.0 cita a expectativa de que outros trabalhos criarão os profiles e extensões necessárias para se alcançar interoperabilidade em uma escala completa.

Dentre os pontos de extensão existentes, dois se destacam a respeito de **grant types** e **token types**. Os **grant types** descrevem as formas que uma aplicação cliente pode obter autorização do usuário para acessar outro sistema em seu nome. A forma apropriada para obter a autorização deve ser escolhida de acordo com o tipo de client.

Os **token types** definem os tipos de token que podem ser usados para se acessar uma aplicação e como se trabalhar com cada tipo. Como exemplos de tipos de token de acesso, existem tokens que não são assinados, que são assinados e tokens autocontidos (mais sobre isso será explicado nos próximos capítulos).

O interessante é que, além dos **grant types** e **tokens types** cobertos pela especificação, é possível criar customizações. Tais customizações podem ser criadas sob demanda para atender apenas um projeto, ou podem ser submetidas para a lista de discussão oauth-ext-review@ietf.org. Os detalhes de como submeter uma extensão podem ser vistos na RFC 6749, em <https://tools.ietf.org/html/rfc6749#section-11>.

1.10 Conclusão

Neste capítulo, apresentei a importância de se utilizar o protocolo OAuth 2.0 considerando o número de aplicações que fornecem APIs e que precisam ser protegidas de diferentes formas. Conhecer a motivação para a criação do protocolo OAuth 2.0 e um pouco da sua história ajuda a entender conceitos indispensáveis para que o desenvolvedor possa encontrar a melhor solução para proteger uma API corretamente.

Nessa altura, você já tem uma ideia sobre como uma API pode ser acessada por aplicações com perfis totalmente diferentes, podendo ser uma aplicação web, um aplicativo nativo ou uma aplicação que execute diretamente no browser (como um aplicativo escrito em JavaScript).

Apesar dos termos apresentados parecerem confusos, pode ficar tranquilo. No próximo capítulo, mostrarei mais detalhes sobre o protocolo antes de começarmos a escrever código.

CAPÍTULO 2

O framework de autorização OAuth 2.0

A essa altura você já conhece um pouco da história do protocolo OAuth 2.0 e da importância da sua utilização para acessar e proteger APIs. Porém, antes de começar a escrever código usando o OAuth 2.0, ainda vamos explorar mais detalhes sobre ele.

É preciso ter em mente algumas definições e entender como as aplicações e o usuário interagem entre si durante os fluxos de autorização e de obtenção de tokens de acesso. O tópico a seguir vai mostrar em detalhes o papel de cada entidade que participa da maioria dos fluxos do OAuth 2.0 e, de forma bem simples, como eles interagem entre si.

2.1 Papéis definidos pelo OAuth 2.0

O protocolo OAuth 2.0 define quatro componentes principais que estão envolvidos na maioria dos fluxos de autorização. Os quatro papéis definidos pela RFC 6749 (<https://tools.ietf.org/html/rfc6749>) são: **Resource Owner**, **Client**, **Resource Server** e **Authorization Server**. Para manter a aderência com a especificação, os nomes serão mantidos em inglês por todo o livro, pois assim você ficará habituado com os termos corretos e não precisará ficar traduzindo mentalmente cada nome sempre que quiser ler outro material em inglês.

A próxima figura mostra os quatro componentes citados e qual o papel de cada um dentro do protocolo OAuth 2.0.

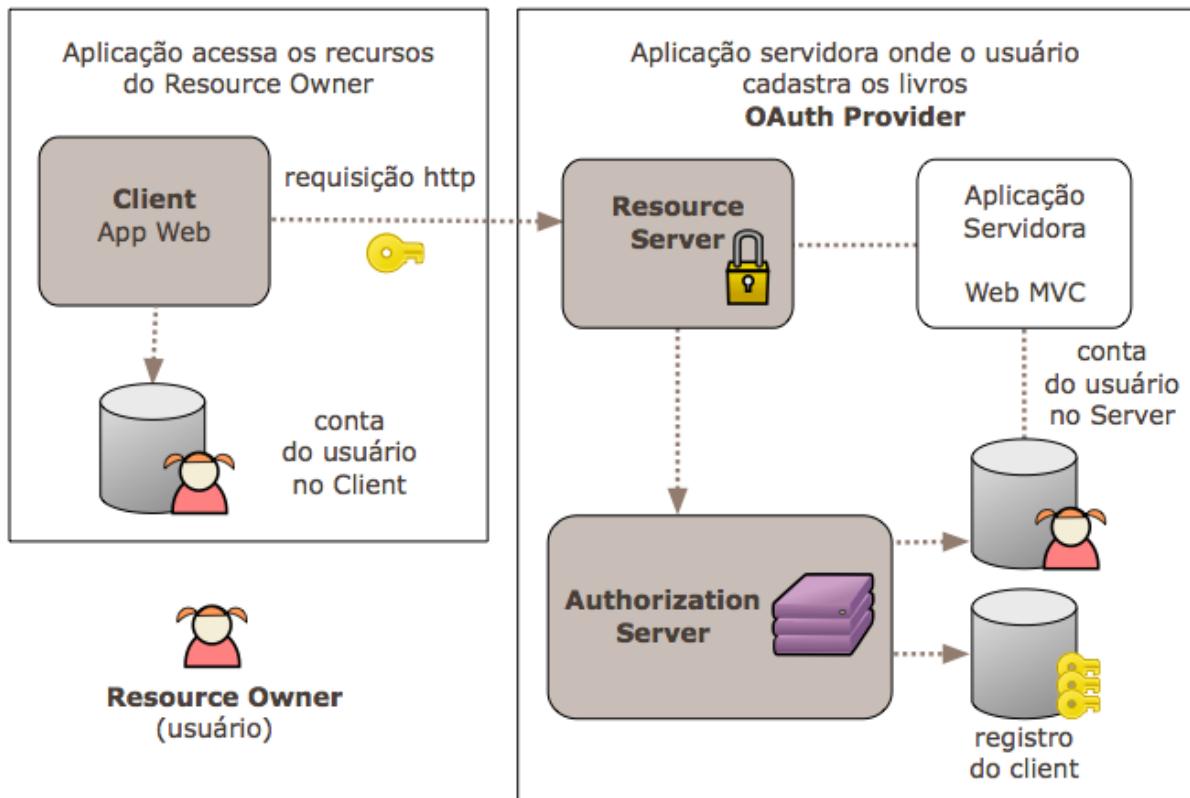


Figura 2.1: Componentes principais do protocolo OAuth 2.0

Além de identificar os papéis definidos dentro do protocolo OAuth 2.0, a figura anterior mostra em linhas gerais como eles se comunicam. Perceba como o Client acessa os recursos do usuário na aplicação servidora utilizando um token de acesso através de uma requisição `http`. Além disso, a figura mostra que o usuário possui conta tanto no Client como na aplicação servidora.

Na figura anterior, a aplicação servidora delimita o espaço dos componentes Authorization Server e Resource Server. Isso identifica a aplicação servidora como um **OAuth Provider**. Alguns exemplos reais de OAuth Provider podem ser conferidos no link <https://oauth.io/providers>, que mostra aplicações como Facebook, Twitter, GitHub, Soundcloud e muitos outros. Vamos agora ver em detalhes como cada um desses componentes interagem entre si.

Resource Owner

A figura anterior mostra o usuário representado como **Resource Owner** (dono dos recursos). Como o Resource Owner interage com cada um dos componentes apresentados na figura? Repare que o desenho do usuário está presente tanto no Client quanto na aplicação servidora.

Para que todo o fluxo do OAuth 2.0 possa acontecer, é preciso que o usuário tenha registro no Client e no OAuth Provider. O registro no OAuth Provider é importante, pois ambos precisam se autenticar. No caso do Resource Owner, este precisa se autenticar antes de autorizar um Client a acessar recursos em seu nome.

Bom, estamos chamando o usuário de Resource Owner (dono dos recursos), mas qual o significado de **recurso** no contexto atual? Como exemplos de recursos, podemos citar uma coleção de livros registrados em um sistema por um usuário, ou até mesmo os dados cadastrais do usuário. Em sistemas que utilizam a arquitetura REST, por exemplo, tais recursos são representados por uma URI, de modo que as operações sobre eles podem ser realizadas através de requisições `http` utilizando-se os métodos `GET`, `POST`, `PUT` e `DELETE`.

Caso queira saber mais sobre a arquitetura REST e como construir APIs utilizando essa abordagem, já existem alguns materiais publicados pela própria *Casa do Código*, como o livro *REST: Construa API's inteligentes de maneira simples*

(<https://www.casadocodigo.com.br/products/livro-rest>). Como a arquitetura REST geralmente é implementada usando o protocolo `http`, vale a pena dar uma conferida na especificação descrita através da RFC 2616. Para mais detalhes sobre os métodos `http` disponíveis, consulte a seção 9 da especificação, em <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9>.

Apesar de o **Resource Owner** ser representado como um usuário, ele também pode ser um sistema. Essa situação existe para atender cenários onde uma aplicação deseja acessar recursos em seu próprio benefício. Pode parecer um pouco estranho uma aplicação precisar utilizar um token de acesso em vez de se autenticar

utilizando login e senha através de *HTTP Basic Authentication*, porém mais adiante esse tipo de cenário será explicado mostrando um bom motivo para uma aplicação se autenticar com um token.

HTTP BASIC AUTHENTICATION

Trata-se de um protocolo de autenticação especificado através da RFC 2617 (<https://www.ietf.org/rfc/rfc2617.txt>), em que um usuário fornece login e senha para acessar um sistema. Através desse protocolo, o login e a senha são concatenados e codificados em Base 64 para poderem ser transmitidos na requisição HTTP. Por conta de serem transmitidos sem criptografia, eles devem ser usados com uma camada de transporte segura através de TLS ou SSL.

Client

A aplicação para a qual o **Resource Owner** concede permissão se trata do **Client**. Como exemplo de **Client**, imagine um aplicativo para celular que publica conteúdo do usuário no Facebook. Fazendo um paralelo com a arquitetura *Client/Server*, o Facebook é o **Server** e o aplicativo é o **Client**.

Considerando a aplicação de livros que vamos usar nos exemplos, o **Client** é quem acessa o sistema de livros para realizar consultas em nome do usuário. Apenas para relembrar, temos os tipos de **Client** públicos e confidenciais.

O **Client** interage com todos os participantes definidos no protocolo OAuth 2.0. Ele interage com o **Resource Server** quando tenta fazer uma requisição `http` para acessar algum recurso do **Resource Owner**. O **Client** também interage com o **Authorization Server** quando solicita a autorização do usuário para acessar os recursos.

Além disso, a comunicação com o **Authorization Server** também é realizada quando o **Client** solicita o token de acesso. Na figura

anterior, perceba também que o desenho que representa o banco de dados do **Authorization Server** possui as chaves geradas para o **Client** e seus dados cadastrais.

Resource Server e o Authorization Server

O **Resource Server** e o **Authorization Server** juntos dão origem ao que estamos chamando de **OAuth Provider**. São eles que, quando implementados, fornecem o suporte ao protocolo OAuth 2.0 para uma aplicação servidora.

O **Resource Server** é servidor responsável por validar se o Client que está tentando acessar os recursos do usuário tem ou não as permissões necessárias. Para isso, ele precisa validar o token de acesso fornecido pelo Client, podendo interagir com o Authorization Server ao acessar um banco de dados compartilhado ou consultar detalhes do token através de algum serviço fornecido pelo Authorization Server. Também é possível utilizar tokens autocontidos que evitam que o Resource Server precise se comunicar com o Authorization Server para validar um token (isso será melhor abordado no capítulo sobre validação de tokens).

Agora falando sobre o **Authorization Server**, uma das principais responsabilidades desse importante componente do protocolo OAuth 2.0 é disponibilizar os tokens de acesso para o Client. Lembre-se de que não queremos que o Resource Owner tenha de gerenciar e entregar tokens de acesso para aplicações terceiras. Para que o Client possa solicitar um token de acesso, primeiramente é necessário que ele tenha feito seu registro no Authorization Server.

Se você já tentou criar um aplicativo que integra com o Facebook, deve ter se deparado com um cadastro no qual era preciso informar alguns detalhes sobre seu projeto. Esse processo se trata justamente do **registro do Client** no Authorization Server. Uma vez que o Client tenha a autorização do Resource Owner para acessar a aplicação em seu nome e tenha obtido um token de acesso junto ao

Authorization Server, ele já pode utilizar o token para fazer uso dos recursos em nome do usuário.

Ao implementar um OAuth Provider, podemos distribuir as responsabilidades de Authorization Server e Resource Server de duas formas: logicamente ou fisicamente.

Ao separar os componentes de um OAuth Provider logicamente, estamos considerando que o Authorization Server e o Resource Server fazem parte de uma mesma aplicação. Essa solução funciona perfeitamente quando usamos OAuth 2.0 para proteger recursos mantidos em apenas uma aplicação, com algumas ressalvas quando o assunto é segurança.

Quando separamos fisicamente o Authorization Server e o Resource Server, tornamos possível utilizar o Authorization Server como um componente que pode gerenciar tokens de acesso para mais do que um Resource Server. Além disso, ao separar em aplicações distintas, também ganhamos segurança. Na separação física, não deixamos todos os ovos na mesma cesta, reduzindo com isso a superfície de ataque do OAuth Provider.

2.2 Registro do Client no Authorization Server

Para que o Client possa iniciar uma solicitação de token de acesso, ele deve ser reconhecido pelo Authorization Server, sendo necessário realizar um registro no servidor de autorização. A forma como esse registro é feito não é coberta pela especificação do OAuth 2.0, deixando em aberto como o Authorization Server deve implementar.

Geralmente, o Authorization Server fornece uma tela para registro de aplicativos que se integram utilizando as APIs disponíveis. Algumas vezes esse processo é realizado manualmente pelo

desenvolvedor do Client (dependendo do fluxo de obtenção de token de acesso usado).

Conforme mostrado na figura a seguir, durante o registro do Client, é **recomendado** informar uma URI de redirecionamento e o tipo de aplicativo. A URI de redirecionamento (ou campo *Redirection URI*) indica para onde o servidor deve devolver um código de autorização quando o Resource Owner ceder permissão de acesso aos seus recursos. O tipo de aplicativo (ou campo *Client Type*) pode indicar se o aplicativo sendo registrado se trata de um sistema web ou mobile.

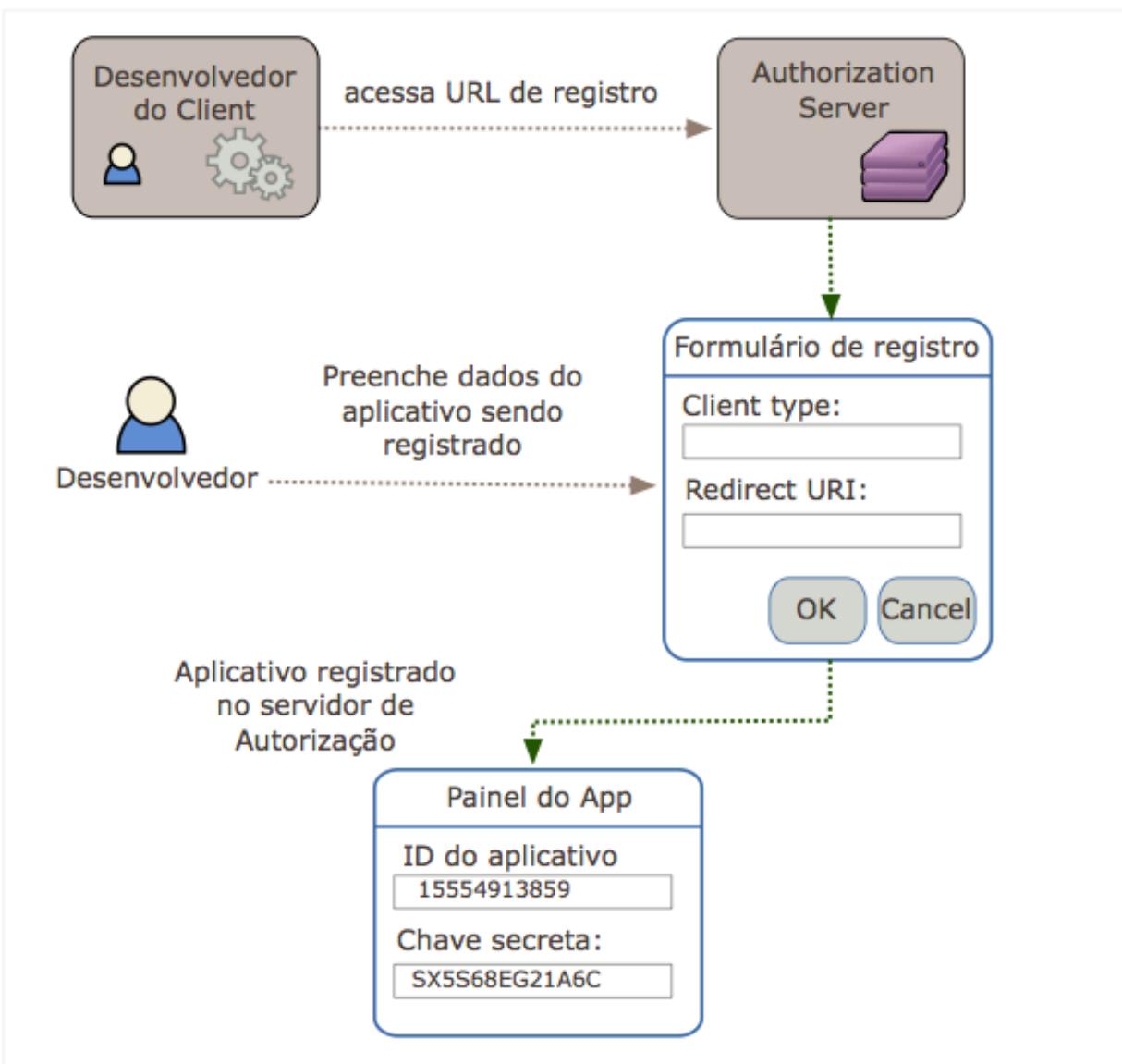


Figura 2.2: Registro do client no servidor de autorização

Por enquanto, não se preocupe como funcionam os processos de autorização e de entrega do token de acesso, pois esses detalhes serão explicados melhor logo adiante. Agora apenas atente-se à importância da utilização do campo `redirect URI` (URI de redirecionamento). Caso o Authorization Server não obrigue o Client a registrar uma URI de redirecionamento, durante o fluxo de autorização é possível que uma URI de um sistema mal-intencionado seja injetada indevidamente e o Authorization Server acabe entregando um token de acesso para a aplicação errada.

Ainda hoje, existem empresas que não se atentam para tais detalhes e acabam deixando brechas de segurança em seus sistemas. Em 28 de Novembro de 2016, um post no site *Threatpost* (BROOK, 2016) falou sobre a correção de uma brecha de segurança realizada pelo PayPal que, caso não fosse resolvida, poderia ter permitido o roubo de tokens associados com qualquer aplicativo que fizesse integração utilizando o protocolo OAuth 2.0. Vale a pena dar uma lida nesse post, pois é um caso que está relacionado com a má utilização do campo que define a URI de redirecionamento.

O processo de registro do Client é praticamente unânime nas soluções que utilizam OAuth 2.0, porém a especificação não obriga que um Client seja registrado para poder acessar APIs usando OAuth 2.0. Conforme citado na RFC 6749 (seção 2.4), o protocolo OAuth 2.0 não exclui o uso de Clients não registrados, porém esse tipo de cenário não é coberto pela especificação e também não será abordado no livro.

Autenticação do Client

Uma vez que o Client tenha realizado seu registro, como este passa a ser reconhecido pelo Authorization Server? Para esse propósito, assim que o Client é registrado, o Authorization Server deve gerar um identificador único para a aplicação sendo cadastrada. Além do identificador do Client, o servidor também pode gerar uma chave secreta para que o Client possa se autenticar sempre que houver comunicação com o Authorization Server. O campo identificador e a chave secreta são definidos como `client_id` e `client_secret` pela especificação OAuth 2.0.

Uma questão que vou enfatizar toda vez que falar de autenticação, seja do Client ou do Resource Owner, é sobre a importância de proteger endpoints de autenticação contra ataques de força bruta (ou *brute force attack*). Em um ataque de força bruta, um usuário mal-intencionado pode criar um `script` que tenta todas as possibilidades de senhas com um determinado número de

caracteres (por isso senhas maiores são mais difíceis de se descobrir).

Existem algoritmos de força bruta que utilizam dicionários com senhas comuns para diminuir o número de tentativas. Além de possibilitar o roubo de credenciais, esse tipo de ataque pode tirar um sistema do ar por conta do número de requisições por segundo enviadas para o Authorization Server. Para essa situação, temos como exemplo de solução o uso de `rate limit`, `Captcha` ou configurar o número de tentativas erradas que podem ser feitas durante a autenticação.

A técnica de `rate limit` permite definir o número máximo de requisições que são aceitas em um determinado intervalo de tempo. E a técnica de `Captcha` é usada em formulários de autenticação nos quais o usuário tenta provar que não é um robô podendo, para isso, descrever o que está vendo em imagens apresentadas na tela.

2.3 Grant types

Conforme dito no capítulo anterior, os **grant types** definem as diferentes **formas de se obter um token de acesso** junto ao Authorization Server. Cada tipo de cliente precisa de uma forma diferente para obter um token de acesso. Uma aplicação mobile, por exemplo, não consegue (de forma nativa) obter um token de acesso se precisar fazer redirecionamentos entre diferentes aplicações.

Para isso, aplicações mobile podem contar com *web views*, componentes suportados pela maioria das plataformas. Aplicações web podem usar redirecionamentos de forma natural, pois contam com o browser como parte de sua arquitetura. Para dar suporte a diferentes tipos de aplicação, o protocolo OAuth 2.0 apresenta **quatro grant types**:

- Authorization Code;

- Resource Owner Password Credentials;
- Implicit;
- Client Credentials.

Não se preocupe em entender como funciona cada grant type por enquanto, pois cada um deles possui um capítulo dedicado, em que exploraremos os detalhes e como implementar cada um na prática, utilizando a aplicação de livros como exemplo.

2.4 Fluxo básico para obtenção do access token

Com o Client devidamente registrado no servidor, para que ele possa utilizar um token de acesso para interagir com uma API em nome do Resource Owner, é preciso obter a autorização do usuário e o token de acesso que é fornecido pelo Authorization Server. De forma bem simples e abstrata, o fluxo para que o Client possa obter um token de acesso pode ser compreendido através da imagem a seguir, porém tenha em mente que esse fluxo pode ser um pouco diferente para alguns grant types. O fluxo mostrado a seguir **assemelha-se** bastante aos grant types Authorization Code e Implicit.

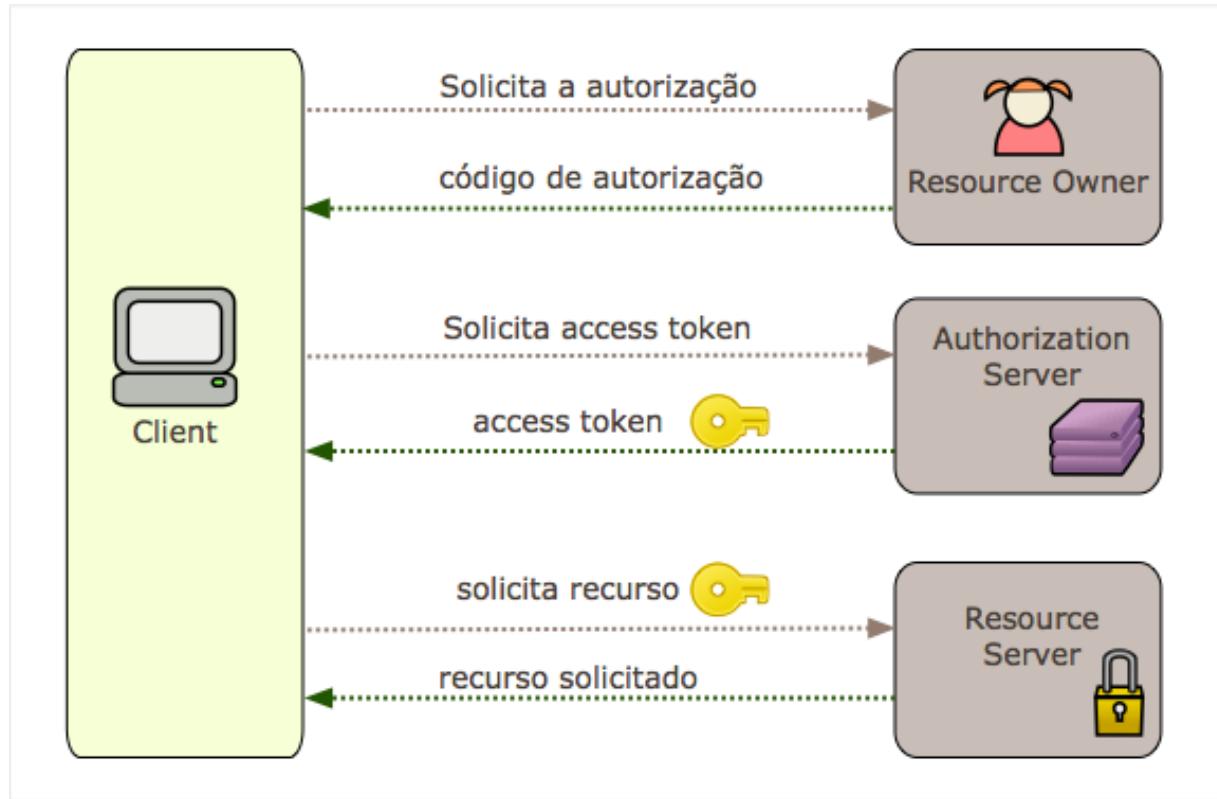


Figura 2.3: Fluxo básico do OAuth

Como podemos ver na figura, o fluxo completo para utilização de token de acesso é composto por três fases: autorização, solicitação do token de acesso e a utilização do token propriamente dito para poder acessar uma API no Resource Server. Se o tipo de aplicação que estivermos considerando for um Client que acessa recursos em seu próprio nome, não faz sentido depender de permissões fornecidas por algum usuário.

Esse tipo de situação será melhor explicado no tópico que detalha o uso de Client credentials grant type. A seguir, serão explicadas cada uma das fases do fluxo do básico do OAuth 2.0.

Autorização

A etapa de autorização se trata do momento em que o usuário efetivamente autoriza um Client a realizar determinadas operações em seu nome na aplicação servidora. O ideal é que a autorização

seja dada de forma indireta para o Client, em que **o usuário se autentica** no Authorization Server e fornece as devidas permissões. Dessa maneira, o Client não fica sabendo nada sobre o login e a senha que ele possui na aplicação protegida por OAuth 2.0.

A autorização só faz sentido para aplicativos que usam os grant types **Authorization Code** ou **Implicit**. O grant type Resource Owner Password Credentials não precisa, porque o Client **repassa** as credenciais do usuário e o grant type Client Credentials não é usado para acessar recursos em nome de um usuário.

A maioria das pessoas que possuem contas em sistemas web hoje em dia já interagiram com um fluxo do OAuth 2.0 ao fornecer permissões para alguma aplicação. Quando você se cadastra em alguma aplicação que realiza integração com o Facebook, por exemplo, geralmente você é redirecionado para uma tela do próprio Facebook, onde faz o login e confirma se deseja compartilhar dados como e-mail, nome e dados de contatos.

Nesse processo, é visível que acontecem **redirecionamentos** entre o Client e o Authorization Server. Primeiro, o Client redireciona o usuário para o Authorization Server. Depois que o usuário fornece as permissões, ele é redirecionado de volta para o Client através da URI de redirecionamento informada no momento do registro do Client.

Junto com essa URI de redirecionamento, o Authorization Server acrescenta um parâmetro obrigatório que representa a autorização (*authorization code*) fornecida pelo Resource Owner para que um Client específico possa acessar um determinado Resource Server. Sendo mais específico, esse parâmetro, chamado `authorization_code` pela especificação OAuth 2.0, deve associar Client, Resource Server e Resource Owner (algo que já é realizado internamente pelo Spring Security OAuth2).

Adicionalmente o `authorization code` deve ser usado apenas uma vez e seu tempo de duração deve ser de, no máximo, 10 minutos

para evitar que outra aplicação instalada na máquina do usuário possa roubar esse código e utilizar para obter um token de acesso.

ATENÇÃO

Não confunda o nome do parâmetro `authorization_code` com o *grant type* Authorization Code. O `authorization_code` é o código retornado pelo Authorization Server, e o *grant type* Authorization Code é o nome de um fluxo de autorização do OAuth 2.0.

Ao redirecionar o usuário para o Authorization Server, o Client também deveria informar mais um parâmetro na URL, chamado `state`, como medida de segurança. A função desse parâmetro será explicada em um capítulo dedicado ao *grant type* Authorization Code.

O fluxo Authorization Code é o mais completo da especificação OAuth 2.0, além de também ser o fluxo que deu origem ao protocolo OAuth. Compreender as etapas de autorização e solicitação do token são importantes para entender os outros fluxos. Os grant types Resource Owner Password Credentials, Implicit e Client Credentials são mais simples, como veremos mais à frente.

Solicitação do token de acesso

A segunda etapa é onde o Client solicita o token de acesso que será usado posteriormente para acessar a API protegida pelo Resource Server. Para solicitar o token de acesso, o Client deve informar o `authorization code` recebido durante a etapa de autorização e, além disso, caso seja um tipo de Client **confidencial**, este **deve se autenticar** no Authorization Server. Se um Client do tipo confidencial não precisasse se autenticar, qualquer aplicação que roubasse um `authorization code` válido poderia facilmente obter um token de acesso para acessar recursos do usuário como se fosse o Client verdadeiro.

Duas questões precisam ser consideradas a respeito de qualquer endpoint que utilize autenticação assim como o endpoint de solicitação de token de acesso. Primeiro, caso seja gerada uma senha para o Client durante o processo de registro, o Authorization Server deve suportar *HTTP Basic Authentication* conforme descrito na RFC 6749 seção 2.3.1. Vale lembrar de que o Authorization Server também pode suportar outros métodos de autenticação *HTTP*, conforme descrito na seção 2.3.2 da especificação do OAuth 2.0.

Segundo, esse tipo de endpoint precisa estar protegido contra ataques de força bruta, conforme já falei anteriormente.

2.5 Utilização do token de acesso

A última etapa do fluxo considera o uso do token de acesso para realizar operações na aplicação em nome do Resource Owner. Uma vez que o Client já possua o token de acesso, ele pode utilizar as APIs do sistema que são protegidas pelo Resource Server. Para usar o token de acesso, o Client precisa apresentar o token para o Resource Server para poder acessar os recursos do sistema em nome do Resource Owner. O token não deve ser enviado para o Authorization Server.

Token types

No primeiro capítulo, falamos que um token de acesso nada mais é do que uma string com um tamanho específico. O Client não deveria se preocupar com o tamanho da string, porém o Authorization Server pode documentar o tamanho do token de acesso que será fornecido. A questão é que existem diferentes tipos de token de acesso.

Um tipo de token muito usado é o **Bearer Token** que, segundo a própria especificação, qualquer portador (daí o nome *bearer*) do

token pode acessar recursos do Resource Owner que autorizou a sua geração. Podemos fazer uma analogia com o dinheiro. Caso uma pessoa encontre uma nota de 100 reais no chão, ela pode usar esse dinheiro em qualquer lugar, pois não será questionada sobre a origem dele. Já um cartão de débito não pode ser usado por outra pessoa (a menos que se descubra a senha do cartão).

Como analogia ao cartão de débito, também existe um tipo de token conhecido como **MAC Token**. Além disso, existem tokens que são codificados com dados adicionais que podem ser usados pelo Resource Server durante o processo de validação do token. Esses tipos de tokens também são conhecidos como tokens autocontidos pois, a partir deles, é possível extrair metadados sem precisar realizar busca em algum banco de dados ou em outra aplicação. Graças à capacidade de extensibilidade do protocolo OAuth 2.0, outros tipos de tokens podem ser criados para suportar futuras novas arquiteturas de sistema que venham a surgir.

Envio do token via HTTP

Agora, como podemos enviar o token de acesso para acessar uma API protegida por OAuth 2.0? Como estamos considerando o protocolo HTTP como base para a comunicação entre as aplicações, o token deve ser enviado através dos mecanismos oferecidos pelo protocolo HTTP. Os mecanismos que podem ser usados para enviar o token de acesso do tipo *Bearer Token* na requisição são:

- Utilizar o Header Authorization do HTTP;
- Enviar o token como um parâmetro de um formulário;
- Enviar o token como um parâmetro de URI.

As formas como um *bearer token* pode ser usado estão especificadas na RFC 6750, que pode ser acessada em <https://tools.ietf.org/html/rfc6750#section-2>.

Token de acesso via parâmetro de URI

Essa é uma das formas mais simples para se enviar um token de acesso, porém um pouco perigosa. Enviar o token de acesso através de parâmetro de `URI` pode não ser uma boa opção, pois o sistema protegido por OAuth 2.0 pode fazer o log de todas as requisições que chegam no servidor. Dessa forma, o token de acesso poderia ser lido por qualquer um que acessasse os logs do sistema. Adicionar o token como um parâmetro da `URI` é bem simples, conforme mostrado a seguir no trecho de comandos `HTTP`.

```
GET /livros?access_token=uAmsMHDpX9TpHvYQW0s1CG12f0FvLi9N HTTP/1.1
Host: localhost:8080
```

Por conta da fragilidade desse método de autenticação, essa forma de utilização do token de acesso só deve ser realizada quando não for possível enviar o token através do header `Authorization` da requisição `HTTP`.

Token de acesso via formulário `HTTP`

Para enviar o token de acesso dessa forma, o Client precisa adicionar o token no corpo da requisição `HTTP` pelo parâmetro `access_token`. Para enviar um token de acesso utilizando formulário, é necessário que a requisição use o `Content-Type application/x-www-form-urlencoded` conforme mostrado a seguir.

```
POST /livros HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded

access_token=uAmsMHDpX9TpHvYQW0s1CG12f0FvLi9N
```

Veja que, no trecho de comandos `HTTP` mostrado anteriormente, usamos o método `POST` do `HTTP` para acessar o endpoint `http://localhost:8080/livros`. Usar essa abordagem para acessar recursos protegidos por OAuth 2.0 pode trazer algumas limitações, pois assim todos endpoints precisam ser acessados utilizando método `POST`. E você não vai querer criar um endpoint que retorne

uma lista de livros usando o método `POST`. O mais óbvio nesse caso seria utilizar o método `GET` do protocolo `HTTP`.

Utilização do Header Authorization HTTP

Para enviar um token de acesso através do cabeçalho `Authorization` do `HTTP`, o Client precisa usar o esquema `Bearer` conforme mostrado no trecho de comandos `HTTP` a seguir:

```
GET /livros HTTP/1.1
Host: localhost:8080
Authorization: Bearer uAmsMHDpX9TpHvYQW0s1CG12f0FvLi9N
```

Essa é a forma mais usada e recomendada para se enviar um token para ser validado pelo Resource Server, pois é mais simples do que enviar como um campo de formulário, e mais seguro do que enviar como um parâmetro de URI.

Refresh tokens

Os tokens de acesso podem ter seu tempo de vida limitado como forma de segurança. Imagine que um Client possua falhas de segurança que comprometam a proteção dos seus tokens de acesso. Uma boa notícia é que outros Clients não serão comprometidos por causa da falha de outro Client.

Entretanto, se um usuário mal-intencionado roubar um token de um Client qualquer, esse usuário poderá realizar operações em nome do Resource Owner por tempo indeterminado. Caso o Authorization Server definir um tempo de validade para o token ao gerar um token de acesso, ele não poderá mais ser usado por tempo indeterminado.

Mas ainda assim essa solução compromete a experiência do usuário. Toda vez que o token expirar, o Client precisará solicitar a autorização do Resource Owner novamente, para em seguida solicitar um novo token de acesso para o Authorization Server. Para resolver esse problema, o protocolo OAuth 2.0 define a utilização de **refresh tokens**.

Através de um refresh token, o Client pode solicitar um novo token de acesso para o Authorization Server, sem a necessidade de ter de incomodar o Resource Owner solicitando sua aprovação novamente. Você poderá ver mais detalhes sobre Refresh Tokens mais adiante.

2.6 Autenticação ou autorização

Uma questão que costuma gerar confusão quando o assunto é o protocolo OAuth 2.0, e que pode até mesmo trazer problemas para as aplicações caso seja mal compreendida, é o fato de o protocolo OAuth 2.0 não ser um protocolo de autenticação.

AUTENTICAÇÃO VERSUS AUTORIZAÇÃO

Autenticação significa identificar que um usuário é mesmo quem diz ser utilizando para isso meios que comprovem sua identidade (geralmente um login e senha, ou uma digital, são suficientes para provar a identidade de um humano).

Autorização, por sua vez, significa identificar se um determinado usuário possui permissão ou não para acessar algo ou para modificar algum conteúdo.

Por ser usado dentro de protocolos de autenticação, muitos desenvolvedores acabam acreditando que podem utilizar OAuth 2.0 para autenticar aplicações clientes. OAuth não diz nada sobre o Client e nem sobre o usuário que autorizou o uso do token de acesso. Inclusive, uma vez que o token de acesso seja gerado, o usuário não precisa estar autenticado na aplicação servidora para que o Client possa acessar o Resource Server.

Se um token de acesso não permite identificar se quem o está usando é realmente o Client correto, essa solução definitivamente

não pode ser considerada como autenticação. O Spring Security OAuth2 até mantém uma associação com o Client dono do token, mas isso não garante que o token foi passado pelo verdadeiro dono.

Caso queira saber mais sobre esse detalhe, vale a pena dar uma lida em um post muito legal do site do oauth.net, disponível em <https://oauth.net/articles/authentication>.

2.7 Conclusão

A essa altura você já conhece os principais componentes da especificação OAuth 2.0 e os princípios básicos do protocolo. Neste capítulo, apresentei os papéis definidos na especificação e os quatro fluxos para obtenção de token de acesso, também conhecidos como *grant type*. Os grant types foram apresentados de maneira bem superficial com um foco maior no Authorization Code, que pode ser considerado um dos fluxos mais conhecidos e completos do protocolo OAuth 2.0.

Agora que você já está familiarizado com os termos usados na especificação e já tem uma noção geral do protocolo, nos próximos capítulos aprofundaremos nos conceitos ao mesmo tempo em que mostrarei na prática como configurar cada um dos componentes de um sistema protegido por OAuth 2.0.

CAPÍTULO 3

Iniciando o projeto de exemplo

A partir de agora, vamos começar a trabalhar na aplicação web de exemplo que batizei de **bookserver**. A ideia dessa aplicação é permitir que os usuários desse sistema possam cadastrar e avaliar os livros que já leram atribuindo notas para eles. Para que o usuário possa interagir com o sistema, já foram disponibilizadas algumas telas com as principais funcionalidades, que são:

- Tela de login;
- Registro de usuários;
- Cadastro de livros;
- Consulta de livros.

A partir da aplicação de exemplo, vamos adicionar algumas funcionalidades na aplicação **bookserver** para permitir a integração com outros sistemas. A integração será realizada através de uma API que disponibilizará *endpoints* para consulta de livros do usuário da aplicação.

Como o usuário precisa permitir o uso de seus dados, este precisa de alguma forma aprovar o acesso aos recursos delegando autorização de acesso para outra aplicação. Inicialmente, os endpoints da aplicação serão protegidos de forma bem simples, com o Spring Security.

Para acompanhar a evolução da aplicação **bookserver**, recomendo que você baixe o projeto que está disponível no GitHub, em <https://github.com/adolfoweloy/livro-spring-oauth2>. Se você tiver o Git instalado em sua máquina, basta executar o comando a seguir e os exemplos serão baixados em um diretório `livro-spring-oauth2`.

```
git clone git@github.com:adolfoweloy/livro-spring-oauth2.git
```

Se não tiver o Git instalado em sua máquina, você também pode baixar um arquivo `zip` do projeto que também está disponível na página do projeto no GitHub.

Caso você não tenha acesso a um computador enquanto estiver lendo, recomendo que se atente para os conceitos e pratique mais tarde assim que puder. A prática é bastante importante e vai ajudar a compreender os fluxos definidos no protocolo OAuth 2.0.

3.1 Estrutura básica do projeto

O projeto disponível no GitHub com o nome `livro-spring-oauth2` contém alguns subdiretórios agrupados por capítulo que contém os exemplos que vamos usar ao longo do livro. Dentro de cada diretório, disponibilizei projetos necessários para cada cenário que for abordado nos respectivos capítulos.

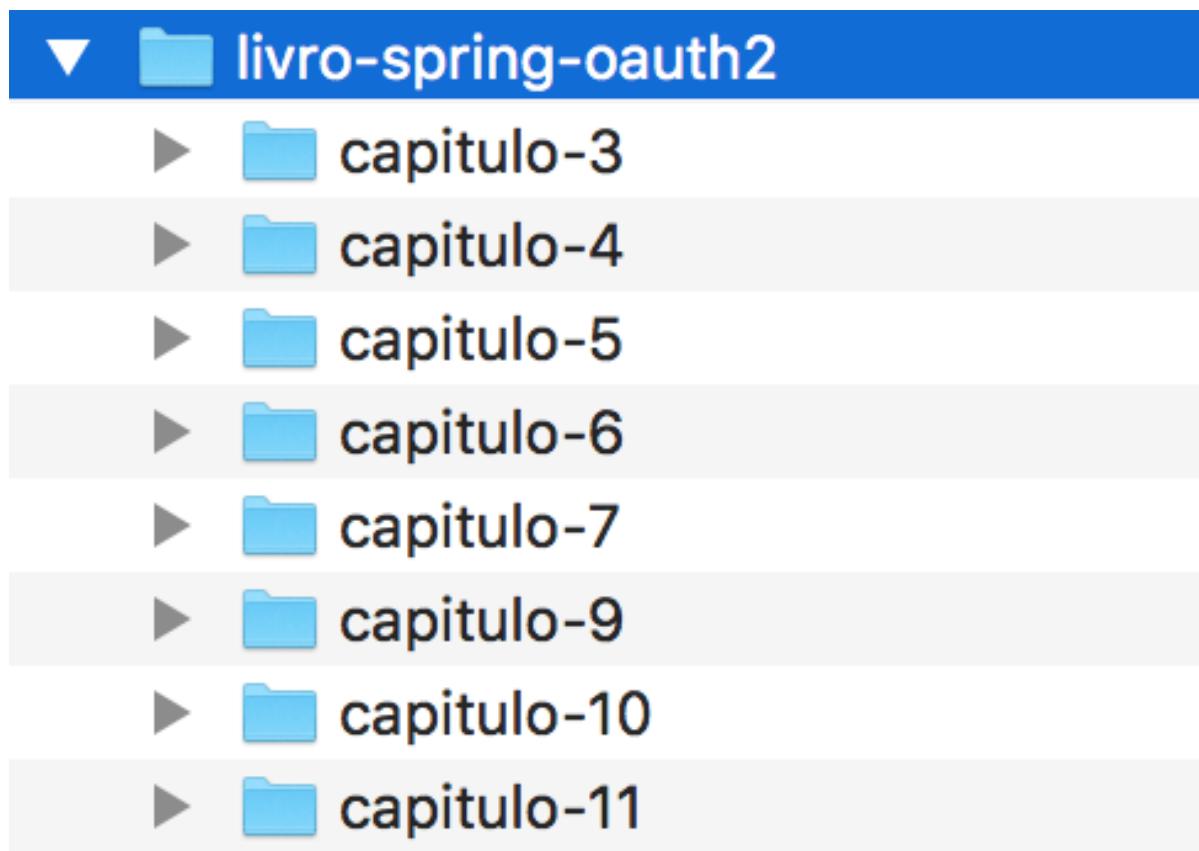


Figura 3.1: Estrutura do diretório de exemplos

Além dos exemplos fornecidos, também disponibilizei para cada capítulo um arquivo contendo os comandos utilizados nos testes seguindo a sequência em que eles aparecem no livro. Esses comandos estão disponíveis nos arquivos de nome `comandos.md`, que acompanham cada diretório com exemplos.

Agora trabalharemos com o primeiro exemplo que deve estar no diretório `livro-spring-oauth2/capítulo-3`. Entre nesse diretório e importe para sua IDE favorita o projeto **bookserver** como um projeto *Maven*. O projeto foi configurado usando o *Maven* para facilitar o gerenciamento das dependências e a execução da aplicação. Assim que a importação do projeto for finalizada, você deve ter acesso à seguinte estrutura:

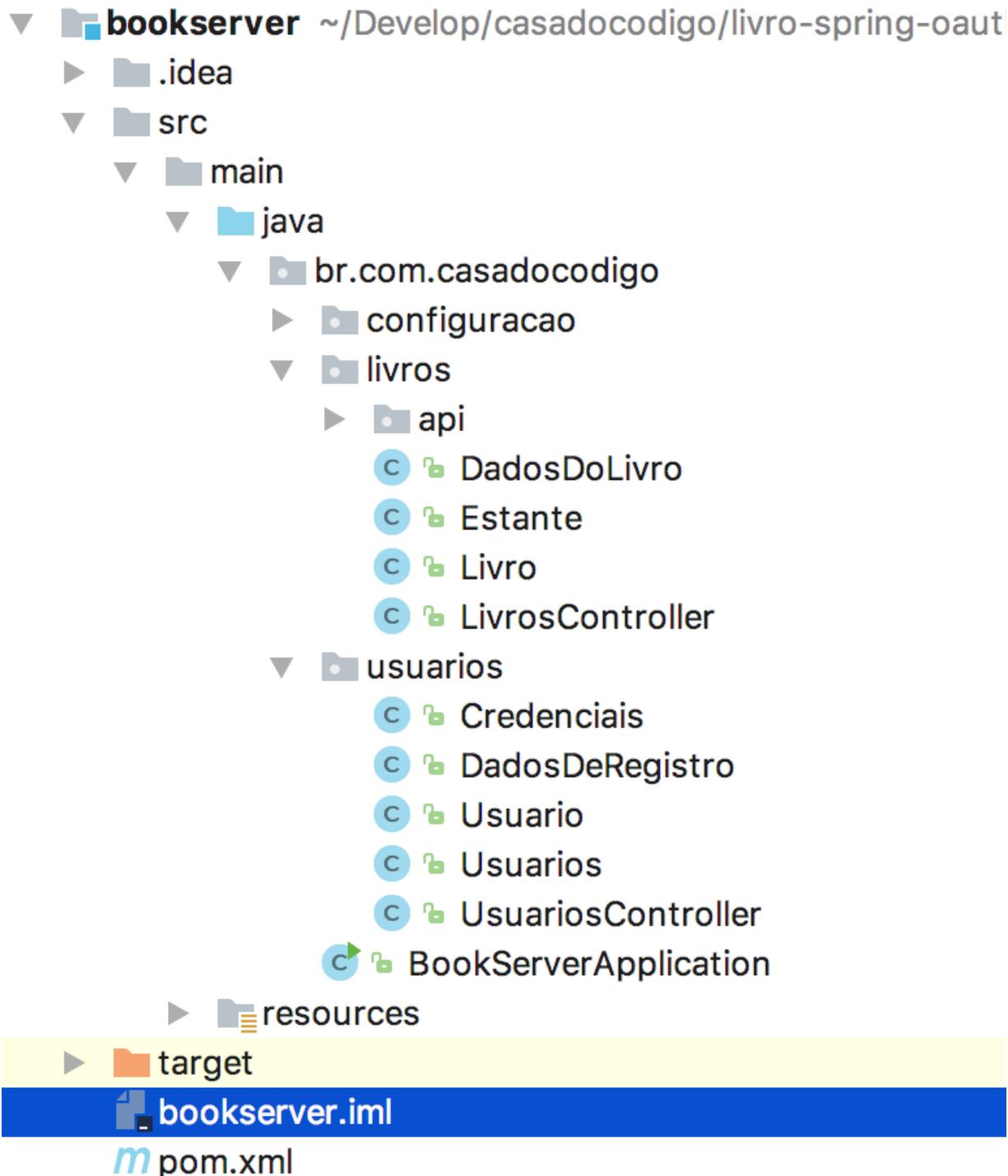


Figura 3.2: Estrutura do projeto bookserver

Abra o arquivo `pom.xml` para entendermos um pouco das dependências sendo usadas no projeto. Perceba que, além das dependências mostradas no trecho de código a seguir, também

foram adicionadas algumas dependências de *template* para renderização das telas do sistema. Mas não se preocupe com elas, pois o foco agora é conhecer as dependências que estão sendo utilizadas para criar e proteger a API do sistema.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.3.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
</dependencies>
```

Repare no arquivo `pom.xml` que a primeira coisa que está sendo referenciada é um projeto `parent` do Spring Boot. Ao usar esse `parent`, o gerenciamento das versões de algumas dependências já é realizado automaticamente, ou seja, não precisamos nem declarar as versões das dependências. Todas já estão declaradas em um arquivo `pom.xml` especial do próprio Spring Boot.

Analizando o trecho do arquivo `pom.xml` mostrado anteriormente, podemos notar que o projeto `bookserver` possui dependências de alguns projetos do Spring, do conector do MySQL e do projeto `lombok`. Agora vamos entender um pouco mais sobre cada uma dessas dependências e por que elas serão úteis para o estudo do OAuth 2.0 usando o Spring Security OAuth2.

3.2 O projeto Lombok

Quando escrevemos as classes que representam as entidades da aplicação, geralmente precisamos definir as propriedades através de métodos de acesso e de atribuição, ou seja, por meio de `getters` e `setters`. Quando criamos muitos desses métodos em nossas classes, elas acabam perdendo um pouco da legibilidade.

Lembre-se de que a criação de muitos `getters` e `setters` no projeto pode ser sinal de utilização de modelos anêmicos. Entretanto, como este é um livro sobre OAuth 2.0, não vou entrar no mérito de design de código utilizando modelos ricos. Mas caso você tenha curiosidade sobre o assunto, recomendo a leitura do livro **Domain Driven Design** do Eric Evans, e o livro **Implementing Domain-Driven Design** do autor Vaughn Vernon.

Design de código à parte, escrever `getters` e `setters` toda hora é chato e, por isso, adicionei o projeto Lombok como dependência do projeto `bookserver`. Além do suporte à geração de `getters` e `setters`, o projeto Lombok também permite a geração automática dos métodos `hashcode`, `equals` e `toString`.

Ao mesmo tempo em que facilita a escrita das classes, as anotações fornecidas pelo Lombok também permitem diminuir a quantidade de código escrito na classe. Em vez de escrever (ou gerar através da IDE) os métodos de acesso de um atributo, basta anotá-los com `@Getter` e `@Setter`. Contudo, para usar o Lombok, não

basta adicionar a dependência no projeto. Também é necessário fazer com que sua IDE favorita passe a reconhecer o Lombok (<https://projectlombok.org/>).

Lombok no Eclipse

Para realizar a configuração do Lombok no Eclipse, baixe o `.jar` do projeto que no momento da escrita deste livro está disponível através do link <https://projectlombok.org/download.html>. Execute o comando a seguir apontando para a localização do arquivo `lombok.jar` na sua máquina (no meu caso, executei o arquivo diretamente do diretório `Downloads`).

```
java -jar ~/Downloads/lombok.jar
```

Ao executar o comando, a seguinte tela deverá aparecer para você fazer a instalação do Lombok em sua IDE. Clique em `install/update` e reinicie a IDE para que ela passe a reconhecer o Lombok.

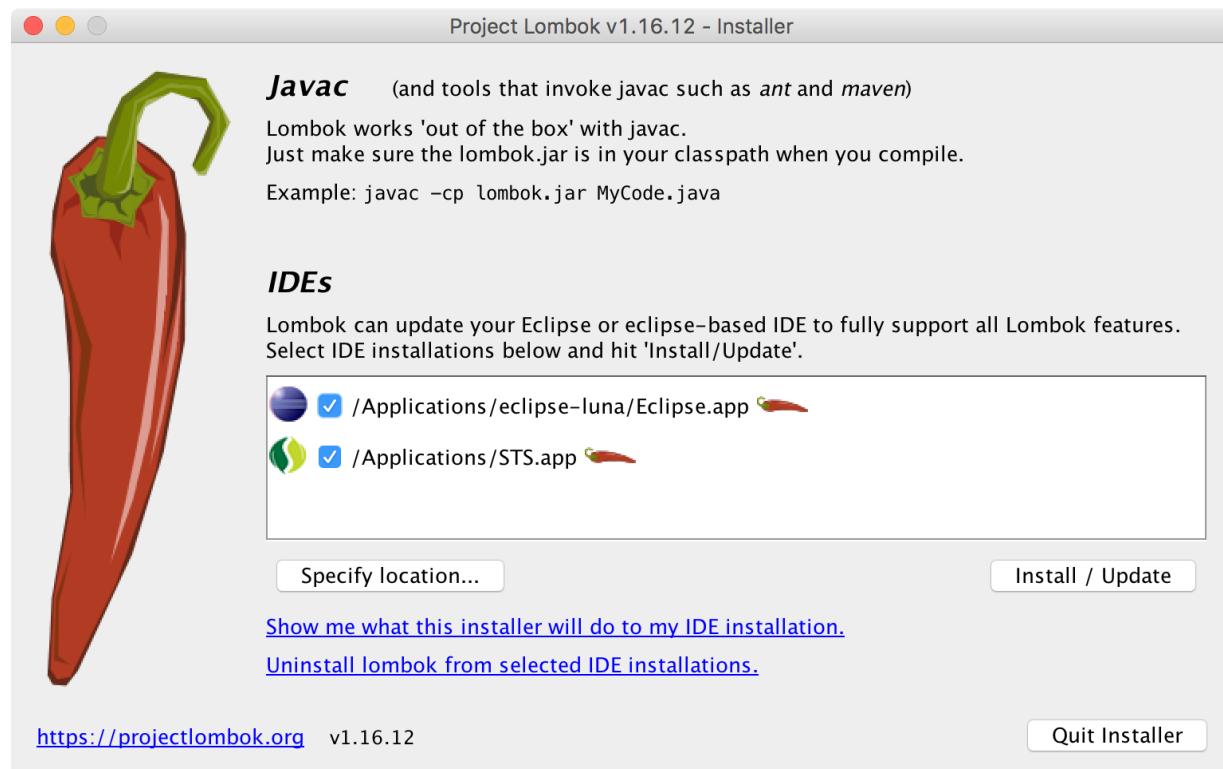


Figura 3.3: Instalação do Lombok

Lombok no IntelliJ

Para configurar o Lombok no IntelliJ, o processo é um pouco mais manual, mas ainda assim bem prático. Baixe o plugin do Lombok disponível na página <https://plugins.jetbrains.com/idea/plugin/6317-lombok-plugin>. Copie o diretório `lombok-plugin` inteiro para dentro de `plugins`, que fica dentro do diretório de instalação do IntelliJ. No meu caso (utilizando Mac OS), copiei o plugin da seguinte forma:

```
cp -r ~/Downloads/lombok-plugin /Applications/IntelliJ\ IDEA\ CE.app/Contents/plugins/.
```

Assim que copiar o diretório de `plugins`, reinicie a IDE e aproveite os benefícios do projeto Lombok. Agora, mesmo não definindo os métodos de acesso, você pode utilizá-los durante a escrita do código.

Veja a seguir a diferença entre uma classe que não usa o Lombok e outra que utiliza para gerar os métodos `equals`, `hashCode` e os getters e setters. A classe `Livro` a seguir não usa o Lombok.

```
public class Livro {  
    private Integer id;  
    private String titulo;  
  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        result = prime * result + ((id == null) ? 0 : id.hashCode());  
        result = prime * result + ((titulo == null) ? 0 :  
        titulo.hashCode());  
        return result;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)
```

```

        return false;
    if (getClass() != obj.getClass())
        return false;
    Livro other = (Livro) obj;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (! id.equals(other.id))
        return false;
    if (titulo == null) {
        if (other.titulo != null)
            return false;
    } else if (! titulo.equals(other.titulo))
        return false;
    return true;
}
}

```

Vejamos agora uma versão bem mais enxuta da classe `Livro` utilizando as anotações do projeto Lombok.

```

@EqualsAndHashCode @ToString
public class Livro {
    @Getter
    private Integer id;

    @Getter
    private String titulo;
}

```

Agora a classe `Livro` possui mais métodos declarados e com menos código. Com isso, os maiores benefícios que podem ser vistos ao utilizar o Lombok são legibilidade e praticidade.

3.3 Spring Boot

Um detalhe importante a se observar no projeto é que estamos utilizando o Spring Boot. O Spring Boot facilita bastante o processo

inicial de criação de uma aplicação bem como sua manutenção durante todo o ciclo de desenvolvimento. Quando vamos criar um projeto com Spring, inicialmente é necessário adicionar uma série de dependências respeitando a compatibilidade entre as versões delas (o que pode ser um pouco trabalhoso dependendo do tamanho do projeto).

Além disso, muitas configurações precisam ser feitas manualmente (seja por meio de configurações Java ou XML, como veremos no próximo capítulo). O Spring Boot facilita todo esse processo de criação e configuração do projeto fornecendo configurações padronizadas que podem ser alteradas conforme a necessidade.

Inclusive, aplicações criadas com o Spring Boot são aplicações *standalone*, ou seja, que podem ser executadas da mesma forma que se executa um `jar`: usando o comando `java -jar app.jar`. Com isso, o desenvolvedor não precisa usar um servidor de aplicação e a realização de deploy do projeto fica muito mais simplificada.

Além das configurações automáticas, o Spring Boot traz o conceito de *starters* que permite adicionar alguns projetos do Spring de forma mais fácil, sem precisar configurar algumas dependências indiretas que podem ser necessárias para o projeto. Os *starters* não são nada mais do que arquivos que descrevem todas as dependências de um projeto do Spring.

Como exemplo de projeto do Spring que é disponibilizado como um *starter*, temos o `spring-boot-starter-data-jpa`. Ao adicionar uma dependência para `spring-boot-starter-data-jpa`, não precisamos adicionar manualmente as dependências para `hibernate`, `jdbc`, `entity-manager` e `transaction-api`, pois o *starter* já faz esse trabalho indiretamente. Além das dependências de banco de dados, os módulos `core` e `aop` do próprio framework Spring também são importados automaticamente.

Existem *starters* para diversos projetos do Spring e, caso queira conhecer mais sobre alguns deles, acesse

<http://docs.spring.io/spring-boot/docs/1.4.3.RELEASE/reference/htmlsingle/#using-boot-starter>.

Quem já trabalhou com o Spring, sem utilizar o Spring Boot, deve lembrar de como era chato ter de adicionar tantas dependências para poder ter uma aplicação web rodando com banco de dados. Com o Spring Boot e o suporte dos *starters* `spring-boot-starter-web`, `spring-boot-starter-data-jpa` e `spring-boot-starter-security`, já temos tudo que precisamos para desenvolver um sistema web com suporte a banco de dados e segurança.

Para facilitar também a inicialização do servidor e a execução dos testes, foi adicionado no arquivo `pom.xml` do projeto, um plugin do Spring Boot para Maven, conforme apresentado a seguir.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Com o plugin `spring-boot-maven-plugin`, podemos iniciar o servidor através de um simples comando. Entre no diretório do projeto `bookserver` e use o seguinte comando pelo terminal:

```
cd livro-spring-oauth2/capitulo-3/bookserver
mvn spring-boot:run
```

Na máquina que utilizei para escrever este livro, o projeto iniciou em 18 segundos e não precisei realizar o deploy em nenhum servidor de aplicação. Mas atenção, pois se você tentar subir a aplicação assim que baixar o projeto, vai se deparar com erro, pois ainda não configuramos o banco de dados da aplicação.

3.4 Configuração do banco de dados

Como a aplicação `bookserver` guarda os livros que um usuário já leu, precisamos de um banco de dados para manter os livros e também o cadastro do usuário. O projeto utiliza o MySQL como banco de dados e, para que a aplicação possa ser executada, foram adicionadas as dependências `mysql-connector-java` e `spring-boot-starter-data-jpa`.

Somente adicionar as dependências não é suficiente para utilizar o banco de dados. Também é preciso realizar a configuração do `data source` que é uma abstração do banco de dados usado pelo Spring Data JPA. Essa configuração foi feita no arquivo de propriedades `application.yml`, conforme mostrado a seguir.

```
spring:
  datasource:
    url: jdbc:mysql://localhost/bookserver
    username: bookserver
    password: 123
    driver-class-name: com.mysql.jdbc.Driver
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL5Dialect
        show_sql: true
      hbm2ddl:
        auto: validate
```

Em minha máquina, criei um banco de dados chamado `bookserver` e defini um usuário com o mesmo nome do banco de dados e uma senha `123`. Essas credenciais não são muito seguras, mas são suficientes para o propósito do livro. Caso seu banco de dados ou as credenciais sejam diferentes, adicione suas configurações nesse arquivo.

Para criar esse banco de dados com as configurações apresentadas, você deve executar os seguintes comandos no

MySQL:

```
CREATE DATABASE bookserver;
CREATE USER 'bookserver'@'localhost' IDENTIFIED BY '123';
GRANT ALL PRIVILEGES ON bookserver.* TO 'bookserver'@'localhost';
```

E para criar as tabelas, accesse o banco de dados através do comando `USE bookserver;`, e em seguida execute os seguintes comandos:

```
-- cria estrutura do banco de dados
create table usuario(
    id int auto_increment primary key,
    nome varchar(100),
    email varchar(100),
    senha varchar(50)
);

create table estante(
    id int auto_increment primary key,
    usuario_id int references usuario(id)
);

create table livro(
    id int auto_increment primary key,
    estante_id int references estante(id),
    titulo varchar(255),
    nota int
);
```

Todos esses comandos também estão disponíveis no arquivo `database.sql`, que está dentro do projeto `bookserver`.

3.5 Modelo de classes do projeto

Antes de criar a API para a aplicação **bookserver**, é necessário conhecer bem o domínio da aplicação. Qual o problema que será resolvido? A ideia do projeto é que um usuário possa se registrar na

aplicação, cadastrar os livros que já leu e adicionar uma nota para o livro.

Pensando no domínio da aplicação, quando uma pessoa começa a colecionar livros, geralmente ela utiliza uma prateleira, ou mesmo uma estante, para guardá-los, logo nada mais apropriado para os nomes das entidades do que `Usuario`, `Estante` e `Livro` conforme mostrado na figura a seguir.

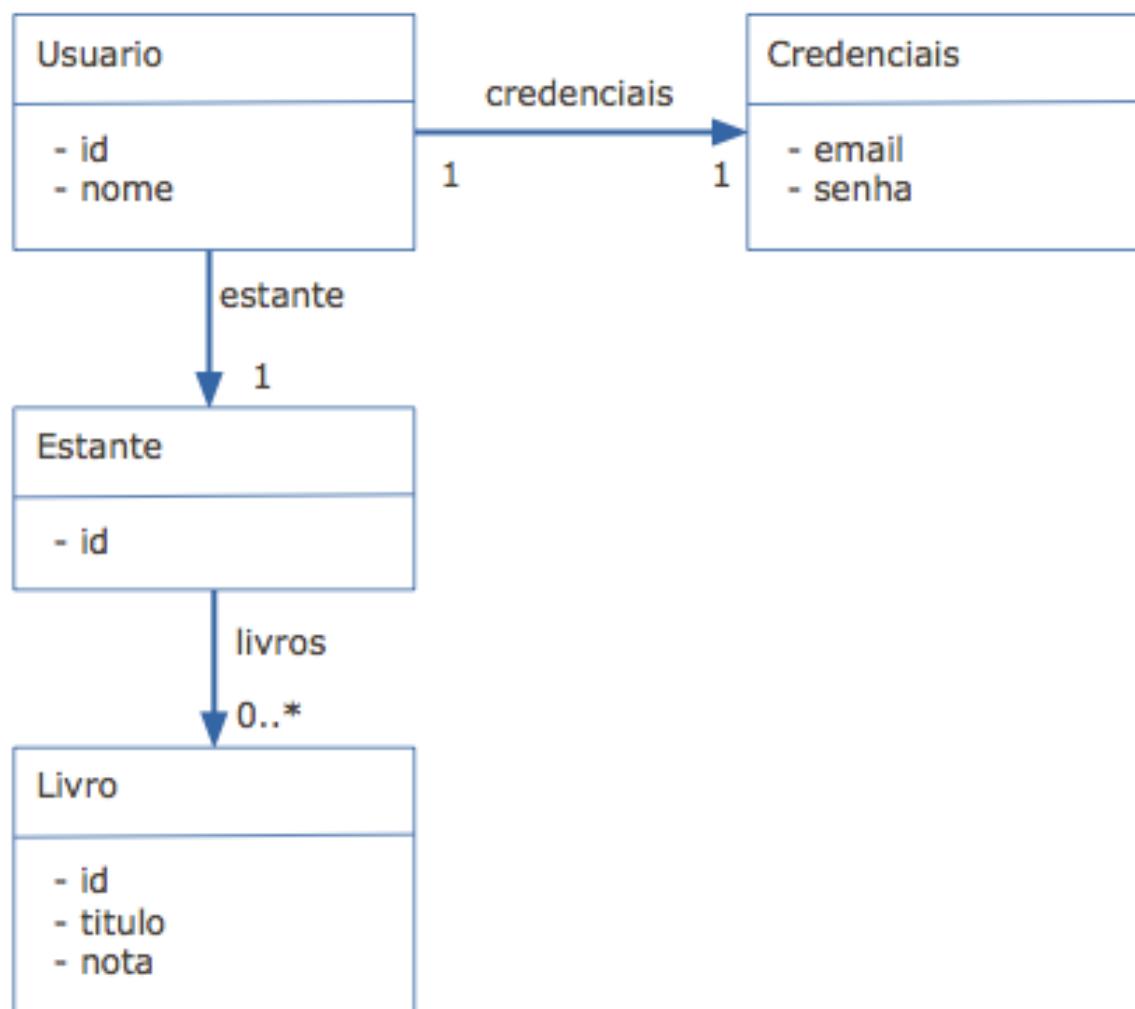


Figura 3.4: Classes de domínio da aplicação

A figura anterior mostra que, além de um `Usuario` ter uma `Estante`, ele também tem credenciais. A classe `Credenciais` representa um

objeto de valor que contém o e-mail e a senha do usuário na aplicação **bookserver**.

Deixando separados esses dados, fica fácil definir qual parte do nosso objeto não deve ser exposta, seja como resposta de uma API ou como saída de log. Para que esses dados não fiquem expostos caso sejam transformados em formato JSON, utilizamos a anotação `@JsonIgnore`.

```
@Entity
public class Usuario {

    @Getter
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Getter
    private String nome;

    @Getter
    @JsonIgnore
    private Credenciais credenciais;

    @Getter
    @OneToOne(mappedBy = "usuario", cascade = { CascadeType.PERSIST,
    CascadeType.MERGE })
    private Estante estante = new Estante();

    @Deprecated
    Usuario() {
    }

    public Usuario(String nome, Credenciais credenciais) {
        super();
        this.nome = nome;
        this.credenciais = credenciais;

        estante.setUsuario(this);
    }
}
```

```
}
```

A classe que contém as credenciais não é uma entidade, mas sim uma classe `Embeddable` conforme mostrado na definição da classe `Credenciais` a seguir.

```
@Embeddable
public class Credenciais {

    @Getter
    private String email;

    @Getter
    private String senha;

    @Deprecated
    Credenciais() {}

    public Credenciais(String email, String senha) {
        super();
        this.email = email;
        this.senha = senha;
    }

}
```

Além da classe com as credenciais do usuário, sempre que o usuário se cadastra no sistema, ele já é criado com uma estante pronta para que ele possa começar a registrar os livros que já leu. A classe `Estante` acaba sendo uma ponte para representar os livros do usuário conforme mostrado no código a seguir.

```
@Entity
public class Estante {

    @Getter
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```

```

@ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
private Usuario usuario;

@OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
@JoinColumn(name = "estante_id")
private List<Livro> livros = new ArrayList<>();

public boolean temLivros() {
    return livros.size() > 0;
}

public Collection<Livro> todosLivros() {
    return Collections.unmodifiableCollection(livros);
}

public void adicionar(Livro livro) {
    livros.add(livro);
}

public void setUsuario(Usuario usuario) {
    this.usuario = usuario;
}
}

```

E a classe que representa os livros ficou definida assim:

```

@Entity
public class Livro {

    @Getter
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Getter
    private String titulo;

    @Getter
    @Range(min = 0, max = 10)
    private int nota;

    @Deprecated
}

```

```
Livro() { }

public Livro(String titulo, int nota) {
    super();
    this.titulo = titulo;
    this.nota = nota;
}

}
```

3.6 Subindo a aplicação

Agora já é possível iniciar a aplicação e realizar alguns testes. Para subir nosso servidor, execute o comando seguinte no diretório do projeto.

```
mvn spring-boot:run
```

Assim que a aplicação tiver iniciado, acesse a URL
<http://localhost:8080> que vai exibir a tela a seguir no seu browser.

Bookserver

Esse é o sistema onde você registra os livros que já leu!!!

Login

Faça login para poder visualizar seus livros e como você classificou cada um.

[Login](#)

Veja os livros que você já cadastrou

Aqui você pode ver os livros que já cadastrou ou adicionar novos livros. Lembre-se que ao registrar um livro você deve informar uma nota!

[Controle de livros](#)

Figura 3.5: Tela inicial da aplicação bookserver

Vamos conhecer um pouco de como a aplicação funciona para o usuário final que também atuará como o **Resource Server** quando for adicionado o suporte ao protocolo OAuth 2.0. Para isso, tente acessar a opção [Controle de Livros](#).

Bookserver

Esse é o sistema onde você registra os livros que já leu!!!

Login

Login:

Senha:

[Enviar](#)

[Voltar](#)

Não tem conta? [Faça seu cadastro](#)

Figura 3.6: Tela de login

Opa, ainda não é possível acessar os dados dos livros. É preciso estar logado na aplicação para começar a cadastrar livros e, como ainda não fizemos um registro na aplicação, vamos cadastrar um usuário acessando o link [Faça seu cadastro](#) no final da página de login.

Bookserver

Cadastre-se

Nome:

E-mail:

Senha:

EnviarCancelar

Figura 3.7: Tela de cadastro

Ao cadastrar uma conta, o usuário já pode começar a cadastrar alguns livros conforme mostrado a seguir.

Bookserver

Conta de oauth@mailinator.com

Controle de livros

Título:

Nota:

[Adicionar livro](#)

Esses são seus livros cadastrados na aplicação bookserver

titulo	nota
Vidas secas	9

[Voltar](#)

Figura 3.8: Tela de cadastro de livros

3.7 A criação da API

Imagine que a aplicação **bookserver** se tornou um sucesso e algumas aplicações terceiras querem se integrar para poder consultar os livros de um determinado usuário. Imagine que o usuário que cadastramos na aplicação também tem uma conta em uma rede social de profissionais e queira permitir a exibição dos livros que ele já leu. Dessa forma, os outros membros da rede social podem saber um pouco mais sobre os interesses do usuário.

Porém, como a aplicação terceira vai consultar os livros de um usuário? Precisamos primeiro criar uma API. Para facilitar, já criei a classe `LivrosApiController` exibida a seguir.

```

@RestController
@RequestMapping("/api/livros")
public class LivrosApiController {

    @Autowired
    private Usuarios usuarios;

    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<?> livros() {

        Estante estante = donoDosLivros().getEstante();

        if (estante.temLivros()) {
            return new ResponseEntity<>(
                estante.todosLivros(), HttpStatus.OK);
        } else {
            return new ResponseEntity<>(
                null, HttpStatus.NOT_FOUND);
        }
    }

    private Usuario donoDosLivros() {
        Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
        ResourceOwner donoDosLivros = (ResourceOwner)
        authentication.getPrincipal();

        return usuarios.buscarPorID(donoDosLivros.getId());
    }
}

```

Repare que, nessa classe, temos o método `livros` que retorna a lista de livros cadastrados pelo Resource Owner. Mas quem será o Resource Owner quando uma aplicação terceira estiver solicitando dados através da API? Inicialmente, será preciso que o Resource Owner passe o login e a senha da aplicação `bookserver` para a aplicação terceira. Vamos então ver um primeiro exemplo de integração utilizando as credenciais do usuário para acessar os recursos na aplicação.

Mas repassar as credenciais do usuário não se trata de um *anti-pattern* que queremos evitar utilizando o protocolo OAuth 2.0? A resposta para essa pergunta é **sim**. Essa forma de integração é mesmo considerada uma prática ruim, porém quero mostrar como a implementação pode ser feita e quais são os problemas práticos dessa abordagem.

A solução inicial para a integração com a aplicação `bookserver` será realizada através de `HTTP Basic Authentication`. Mas vamos ver como proteger o endpoint de consulta de livros com o `Spring Security` no próximo capítulo.

3.8 Conclusão

Neste capítulo, ainda não vimos nada sobre OAuth 2.0 e nem sobre segurança de APIs. Entretanto, já conhecemos o funcionamento da aplicação `bookserver`, quais tecnologias estão sendo usadas e que precisamos adicionar segurança no endpoint de consulta de livros.

Também apresentei aqui alguns conceitos básicos sobre o Spring Boot e como ele ajuda o desenvolvedor a se preocupar com o que realmente importa: com as funcionalidades da aplicação sem precisar perder tempo procurando versões compatíveis de bibliotecas e frameworks para o projeto. Nos capítulos seguintes, você vai acompanhar a evolução da aplicação; vamos adicionar proteção da API com `HTTP Basic Authentication` e depois OAuth 2.0.

CAPÍTULO 4

Proteja a aplicação com Spring Security

Neste capítulo, vamos dar os primeiros passos para proteger a API da aplicação `bookserver` utilizando como exemplo de código o projeto disponível no diretório `livro-spring-oauth2/capitulo-4`. Inicialmente, será apresentado como adicionar segurança utilizando *HTTP Basic Authentication* com o Spring Security.

Nos capítulos seguintes, usaremos o Spring Security OAuth2 que, conforme será mostrado, depende do Spring Security. Por isso é importante entender um pouco sobre o funcionamento básico do Spring Security e sobre sua arquitetura. A arquitetura do Spring Security é bastante flexível, e permite que outras bibliotecas sejam integradas para que sua aplicação forneça diferentes mecanismos de segurança. Durante a configuração do projeto, mostrei que a aplicação possui como dependência o `spring-boot-starter-security`.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Além de adicionar o *starter* do Spring Security, para começar a proteger a aplicação é preciso fazer o seguinte:

- Configurar o projeto para reconhecer o Spring Security e as classes que habilitam o Spring Security durante uma requisição HTTP;
- Configurar usuários e os acessos permitidos;
- Configurar URLs que devem ser protegidas.

Todas essas configurações podem ser realizadas de diferentes maneiras, como por exemplo, através da utilização de um arquivo XML ou através de uma classe de configuração. No projeto `bookserver`, a configuração é realizada pela classe

`ConfiguracaoDeSeguranca`, pois realizar as configurações usando Java em vez de XML traz as seguintes vantagens:

- *Type safety*: escrevendo as configurações diretamente através de código Java, os erros são detectados em tempo de compilação.
- *Code completion*: como a maioria das IDEs dão suporte a completar o código durante o desenvolvimento, fica mais fácil explorar os recursos oferecidos pelo framework durante a configuração. Apesar de algumas IDEs ajudarem no processo de escrita de XML através de *schemas*, buscar por referências para os XMLs dentro da maioria das IDEs é um pouco mais complicado do que simplesmente inspecionar o código do framework.
- *Refactoring*: as IDEs também tornam muito mais fácil o processo de refatoração em código do que em arquivos texto como um XML.

Para ajudar você a tomar a melhor decisão sobre qual abordagem usar em seus próprios projetos, mostrarei a seguir como seria realizada a configuração do Spring Security através de XML antes de mostrar os exemplos de código usando anotações do Java.

4.1 Configuração da aplicação usando XML

Imagine que escolhemos configurar uma aplicação web com Spring Security através de arquivos XML. Ou seja, imagine que essa aplicação utilize arquivo `web.xml` para a aplicação web e um arquivo de configuração do Spring Security também em XML. O código a seguir mostra como seria a configuração da aplicação através do arquivo `web.xml` (não coloquei os namespaces do XML para facilitar a leitura do código).

```
<web-app>
  <display-name>exemplo</display-name>
```

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext-security.xml</param-
value>
    </context-param>

</web-app>
```

O arquivo `web.xml` apresentado contém a configuração do **filtro** `DelegatingFilterProxy` que intercepta as requisições selecionando as que podem acessar recursos protegidos na aplicação. A classe `DelegatingFilterProxy` é considerada um filtro, pois implementa a interface `Filter`.

FILTROS

Filtros (*filters*) são definidos na especificação de Servlets e são usados para adicionar comportamento a qualquer tipo de recurso web. Dessa forma, é possível transformar tanto o conteúdo das requisições como o conteúdo das respostas.

Com filtros, é possível otimizar as respostas compactando o conteúdo a ser retornado. Por exemplo, em vez de retornar imagens com o tamanho original, retornaríamos imagens usando compactação `gzip`.

Dentre as várias utilidades dos filtros, o Spring Security utiliza-os para adicionar uma camada de segurança aos recursos web. Dessa forma, antes de realizar qualquer processamento no servidor é possível configurar padrões de URL que devem ser interceptados de modo a garantir que um recurso não seja acessado por um usuário não autorizado. Caso queira saber mais sobre os filtros, no blog da Caelum há um ótimo material que pode ser acessado através do link

<https://www.caelum.com.br/apostila-java-web/recursos-importantes-filtros/>.

A configuração do filtro define qual é a classe que deve interceptar as requisições para as URLs definidas na tag `filter-mapping`. De acordo com as configurações apresentadas no arquivo `web.xml`, todas as requisições HTTP que chegarem no caminho `/*` serão interceptadas pela classe `DelegatingFilterProxy`, que foi definida com o nome `springSecurityFilterChain`. O nome desse filtro deve ser `springSecurityFilterChain` obrigatoriamente, pois esse mesmo nome é referenciado internamente pelo Spring Security.

A camada de segurança introduzida ao usar o filtro do Spring Security é interessante por construir uma ponte entre um filtro comum da especificação de Servlets e um filtro especial gerenciado pelo contexto de aplicação do Spring. A classe `DelegatingFilterProxy`

não faz todo o trabalho sozinha. Em vez disso, ela delega parte do trabalho para a classe `FilterChainProxy`, conforme mostrado na figura a seguir (internamente essa classe é referenciada como `delegate`).

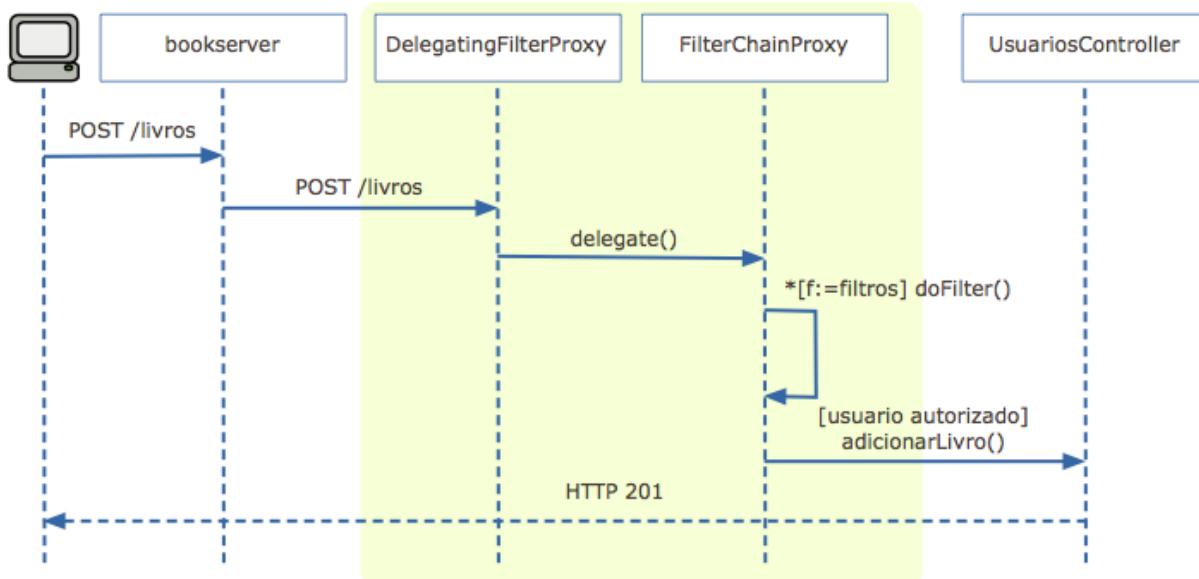


Figura 4.1: Interceptação das requisições HTTP realizadas pelo Spring Framework

Assim como a classe `DelegatingFilterProxy`, a classe `FilterChainProxy` também implementa a interface `Filter` da especificação de Servlets. Entretanto, apesar de `FilterChainProxy` também ser considerada um filtro, essa classe não atua como um filtro qualquer. Esse filtro é especial, pois ele é gerenciado pelo contexto de aplicação do Spring Security. Por conta disso, além de esse objeto possuir a semântica de um filtro, ele é criado como um `bean` do Spring.

Ao declarar uma classe como `bean` do Spring, é possível utilizar recursos de injeção de dependência assim como qualquer outro `bean` gerenciado pelo Spring. O objeto `FilterChainProxy` delega todo o processo de proteção da aplicação para uma cadeia de outros filtros com responsabilidades específicas. Ao adicionar proteção através de *HTTP Basic Authentication*, os filtros mostrados na próxima figura são utilizados em sequência um após o outro.

`WebAsyncManagerIntegrationFilter`

`SecurityContextPersistenceFilter`

`HeaderWriterFilter`

`LogoutFilter`

`BasicAuthenticationFilter`

`RequestCacheAwareFilter`

`SecurityContextHolderAwareRequestFilter`

`AnonymousAuthenticationFilter`

`SessionManagementFilter`

`ExceptionTranslationFilter`

`FilterSecurityInterceptor`

Figura 4.2: Filtros usados em uma URL protegida por Basic Authentication

O último filtro definido através da classe `FilterSecurityInterceptor` (que está em destaque) estende a classe `AbstractSecurityInterceptor` que possui toda a lógica que verifica se um recurso pode ou não ser acessado. A classe `AbstractSecurityInterceptor` é essencial dentro da arquitetura do Spring Security, pois ela colabora com diversas outras classes com responsabilidades bem definidas para realizar autenticação, autorização e muito mais.

Em caso de utilização de arquivo XML, também precisamos indicar como o arquivo de configuração será interpretado e carregado pelo Spring Security. Para isso, é necessário configurar um *listener* do Spring Security na aplicação web.

LISTENERS

A especificação de Servlets fornece suporte para que uma aplicação possa realizar alguma operação quando determinados eventos são lançados em momentos específicos durante a execução da aplicação. Esses eventos podem ser tratados criando *listeners*.

Para criar *listeners*, podemos implementar interfaces como `ServletContextListener`, `ServletRequestListener` e `HttpSessionListener`. Ao implementar uma interface dessas, quando um determinado evento ocorrer (por exemplo, a aplicação acabou de ser inicializada), a aplicação pode "escutar" tal evento e executar uma lógica específica nesse momento.

O *listener* do Spring Security que foi configurado no arquivo `web.xml` se trata da classe `ContextLoaderListener` que implementa `ServletContextListener`. Nesse caso, quando a aplicação web tiver sido inicializada, esse listener vai procurar pelo arquivo de configuração do Spring Security, cujo caminho foi definido através

de um parâmetro de contexto chamado `contextConfigLocation` e cujo valor foi definido como `/WEB-INF/applicationContext-security.xml`.

O listener vai usar o conteúdo do arquivo `applicationContext-security.xml` para criar a cadeia de filtros que adiciona a camada de segurança do Spring Security. O código seguinte mostra como proteger uma URL chamada `/usuarios` utilizando *HTTP Basic Authentication*.

```
<?xml version="1.0" encoding="utf-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-
security.xsd">

    <security:http auto-config="true">
        <security:intercept-url pattern="/usuarios" access="permitAll"/>
        <security:intercept-url pattern="/*" access="isAuthenticated()"/>
        <security:http-basic />
    </security:http>

    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user name="adolfo"
                               authorities="ROLE_MEMBER" password="123" />
                <security:user name="jujuba"
                               authorities="ROLE_MEMBER" password="123" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

Através do arquivo de configuração mostrado como exemplo, foi possível definir quais URLs precisam de autenticação para serem acessadas, qual o tipo de autenticação (nesse caso, *Http Basic*

Authentication), além de definir os usuários que podem acessar a aplicação. Como se pode ver, usar os arquivos XML para configuração exige a escrita de um bocado de código.

Os exemplos com XML são mostrados apenas para que você possa ter uma noção de como funciona essa abordagem de configuração. Isto é, não se preocupe em configurar a aplicação de exemplo desse jeito, pois durante todo o desenvolvimento do projeto bookserver será usada apenas a abordagem através de anotações.

4.2 Configuração efetiva com anotações

Como estamos usando o Spring Boot, muita configuração será feita de forma automática. Precisamos configurar manualmente apenas as URLs que devem ser protegidas e qual o mecanismo de autenticação e autorização a ser utilizado. Toda configuração que apresentei através de arquivo XML também pode ser realizada através de uma classe Java.

Para que o usuário possa se logar no sistema, já foi criada uma classe chamada `ConfiguracaoDeSeguranca.java`. Por enquanto, ela possui uma classe interna chamada `ConfiguracaoParaUsuario` que, por sua vez, contém a configuração necessária para que um usuário final possa se autenticar na aplicação através de uma página de login.

```
@EnableWebSecurity
public class ConfiguracaoDeSeguranca {

    @Configuration
    public static class ConfiguracaoParaUsuario extends
        WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
            String[] caminhosPermitidos = new String[] {
```

```

        "/",
        "/home",
        "/usuarios",
        "/webjars/**",
        "/static/**",
        "/jquery*"
    };

    http
        .authorizeRequests()
            .antMatchers(caminhosPermitidos).permitAll()
            .anyRequest().authenticated().and()
        .formLogin()
            .permitAll().loginPage("/login").and()
        .logout()
            .permitAll().and()
        .csrf().disable();
    }
}
}

```

Antes de configurar a segurança do endpoint de consulta de livros, observe a anotação `@EnableWebSecurity` colocada na classe `ConfiguracaoDeSeguranca`. Essa é uma meta-anotação disponibilizada pelo Spring Security que é composta por outras anotações. Veja a seguir o código-fonte da anotação `@EnableWebSecurity`.

```

@Retention(value = java.lang.annotation.RetentionPolicy.RUNTIME)
@Target(value = { java.lang.annotation.ElementType.TYPE })
@Documented
@Import({ WebSecurityConfiguration.class,
ObjectPostProcessorConfiguration.class, SpringWebMvcImportSelector.class
})
@EnableGlobalAuthentication
@Configuration
public @interface EnableWebSecurity {
    boolean debug() default false;
}

```

Preste atenção nas anotações `@Import` e `@Configuration`. Elas permitem realizar a maioria das configurações do Spring através de código Java. Uma classe anotada com `@Configuration` será processada pelo Spring para criar beans gerenciados pelo contexto do Spring.

É através de classes anotadas com `@Configuration` que podemos realizar todas as configurações do Spring sem precisar definir os objetos em XML através da tag `<bean>`. Veja um exemplo de definição de `bean` no código de uma classe fictícia de configuração chamada `MeusServices`.

```
@Configuration
public class MeusServices {

    @Bean
    public ServicoDeUsuarios servicoDeUsuarios(DataSource dataSource) {
        return new ServicoDeUsuarios(dataSource);
    }

    @Bean
    public DataSource dataSource() {
        // lógica para criação de um datasource
    }
}
```

Na classe de configuração mostrada como exemplo, dois beans estão sendo declarados: `servicoDeUsuarios` e `dataSource`. O nome do método já define o próprio nome do `bean`. Os beans declarados na classe de configuração ficarão disponíveis no contexto do Spring, podendo ser injetados em um `Controller`, por exemplo.

Voltando para o código-fonte da anotação `EnableWebSecurity`, ao ser usada, a anotação `@Import` permite importar configurações definidas em uma outra classe anotada com `@Configuration`. Veja um exemplo de importação de configurações na classe `OutrosServices`.

```
@Import(MeusServices.class)
@Configuration
public class OutrosServices {
    // declarações de beans
}
```

Na classe `OutrosServices`, estamos importando as configurações realizadas em `MeusServices`. No caso da nossa classe de configuração `ConfiguracaoDeSeguranca`, estamos definindo essa classe

como uma de configuração que importa algumas outras configurações, sendo uma das mais importantes a classe `WebSecurityConfiguration`.

A classe `ConfiguracaoDeSeguranca` não possui declarações de `beans`, porém indiretamente isso está sendo feito pela classe `WebSecurityConfiguration`. A classe `WebSecurityConfiguration` declara alguns beans importantes para que a camada de segurança do Spring seja adicionada na aplicação.

Do mesmo modo que o Spring Security precisa das classes `DelegatingFilterProxy` e `FilterChainProxy` explicadas anteriormente quando mostrei as configurações através de XML, o Spring Security também precisa delas quando usamos configuração através de anotações do Java. E um dos `beans` declarados pela classe `WebSecurityConfiguration` é justamente um objeto do tipo `FilterChainProxy`. Vejamos como esse `bean` é declarado através do método `springSecurityFilterChain`.

```
@Bean(name =
AbstractSecurityWebApplicationInitializer.DEFAULT_FILTER_NAME)
public Filter springSecurityFilterChain() throws Exception {
    boolean hasConfigurers = webSecurityConfigurers != null
        && ! webSecurityConfigurers.isEmpty();
    if (! hasConfigurers) {
        WebSecurityConfigurerAdapter adapter = objectPostProcessor
            .postProcess(new WebSecurityConfigurerAdapter() {
            });
        webSecurity.apply(adapter);
    }
    return webSecurity.build();
}
```

É a partir desse método que toda a camada de segurança do Spring será criada. Entretanto, o código mostrado anteriormente parece fazer pouca coisa para quem configura todos os filtros necessários para autenticação, autorização, gerenciamento de sessão, login, logout etc. A questão é que a classe `WebSecurityConfiguration` delega toda a complexidade de criação das configurações para alguns

builders como o objeto `webSecurity`, mostrado no final do método `springSecurityFilterChain`.

A classe `WebSecurity` usada pela `WebSecurityConfiguration` é um `SecurityBuilder` que tem como responsabilidade realizar a construção da camada de segurança, composta por filtros. O próprio **builder** `WebSecurity` utiliza outro **builder** mais especializado, chamado `HttpSecurity`, para criar objetos do tipo `DefaultSecurityFilterChain`.

Agora como os **builders** citados podem criar as configurações de acordo com a necessidade de cada aplicação que criarmos? Para isso, é necessário criar um `WebSecurityConfigurer` e é exatamente isso que foi feito através da classe `ConfiguracaoParaUsuario` de forma indireta. Perceba que essa classe de configuração interna estende `WebSecurityConfigurerAdapter`.

A classe `WebSecurityConfigurerAdapter` implementa `WebSecurityConfigurer` que, por sua vez, estende a interface `SecurityConfigurer`. É através de **configurers** que podemos definir propriedades específicas que vão customizar como as camadas de segurança serão criadas pelos **builders**. Ao estender a classe `WebSecurityConfigurerAdapter`, podemos customizar os **builders** `WebSecurity` e `HttpSecurity` bastando para isso sobrescrever os seguintes métodos:

```
@Override  
public void configure(WebSecurity web) throws Exception {  
}  
  
@Override  
protected void configure(HttpSecurity http) throws Exception {  
}
```

Vamos então adicionar a configuração para o endpoint de consulta da aplicação `bookserver`, criando para isso uma nova classe interna que chamarei de `ConfiguracaoParaAPI`.

```

@EnableWebSecurity
public class ConfiguracaoDeSeguranca {

    @Configuration
    @Order(1)
    public static class ConfiguracaoParaAPI extends
    WebSecurityConfigurerAdapter {

        }

    @Configuration
    public static class ConfiguracaoParaUsuario extends
    WebSecurityConfigurerAdapter {
        // código omitido
    }

}

```

Agora a classe `ConfiguracaoDeSeguranca` possui duas classes internas que serão usadas para configurar a segurança da aplicação `bookserver`. A classe `ConfiguracaoParaAPI` possui mais uma anotação importante: `@Order`. Esta foi adicionada para permitir que as configurações de login via formulário e `HTTP Basic Authentication` possam coexistir. Dessa forma, a aplicação pode ser acessada tanto por um usuário comum como por um outro sistema.

Continuando a configuração para a nossa API, primeiro precisamos configurar quais URLs devem ser protegidas. Adicione o seguinte método dentro da classe `ConfiguracaoParaAPI`.

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .anyRequest().authenticated().and()
        .antMatcher("/api/livros")
            .httpBasic().and()
        .csrf().disable();
}

```

Através do método `configure` adicionado, estamos customizando algumas propriedades do **builder** `HttpSecurity` definindo quais padrões de URL devem ser protegidos, usando autenticação `HTTP Basic Authentication`. Além disso, também estamos desabilitando a proteção contra ataques CSRF, pois as APIs da aplicação bookserver são *stateless*, ou seja, não utilizam o conceito de sessão.

ATAQUE CSRF (CROSS SITE REQUEST FORGERY)

Um ataque CSRF permite que um usuário mal intencionado execute operações em nome de outro usuário, aproveitando-se do contexto de navegação da vítima. Considerando uma aplicação web padrão (com processamentos realizados no servidor e com telas que podem ser usadas pelo usuário), uma das formas de se proteger contra ataque CSRF é fazer com que o servidor crie e devolva para o navegador uma variável com um conteúdo aleatório. Dessa forma, sempre que for realizada uma nova requisição, esse valor aleatório deve ser devolvido para o servidor que, por sua vez, deve verificar se é válido para a sessão atual.

Apesar de mostrar um pouco de como a camada de configuração do Spring Security é criada internamente, ainda existe muito mais acontecendo por trás dos **builders** e **configurers**. Para ajudar a ter uma visão de alto nível do fluxo de configuração da camada de segurança, veja a figura a seguir que ilustra como o `FilterChainProxy` é criado.

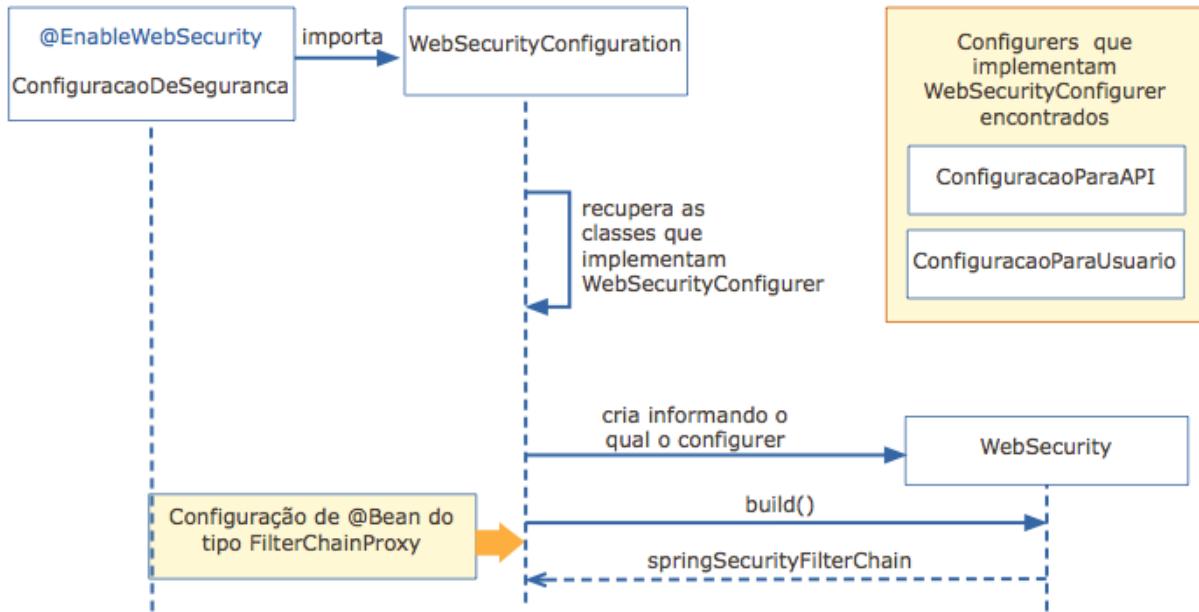


Figura 4.3: Criação do `FilterChainProxy` através da configuração por anotações

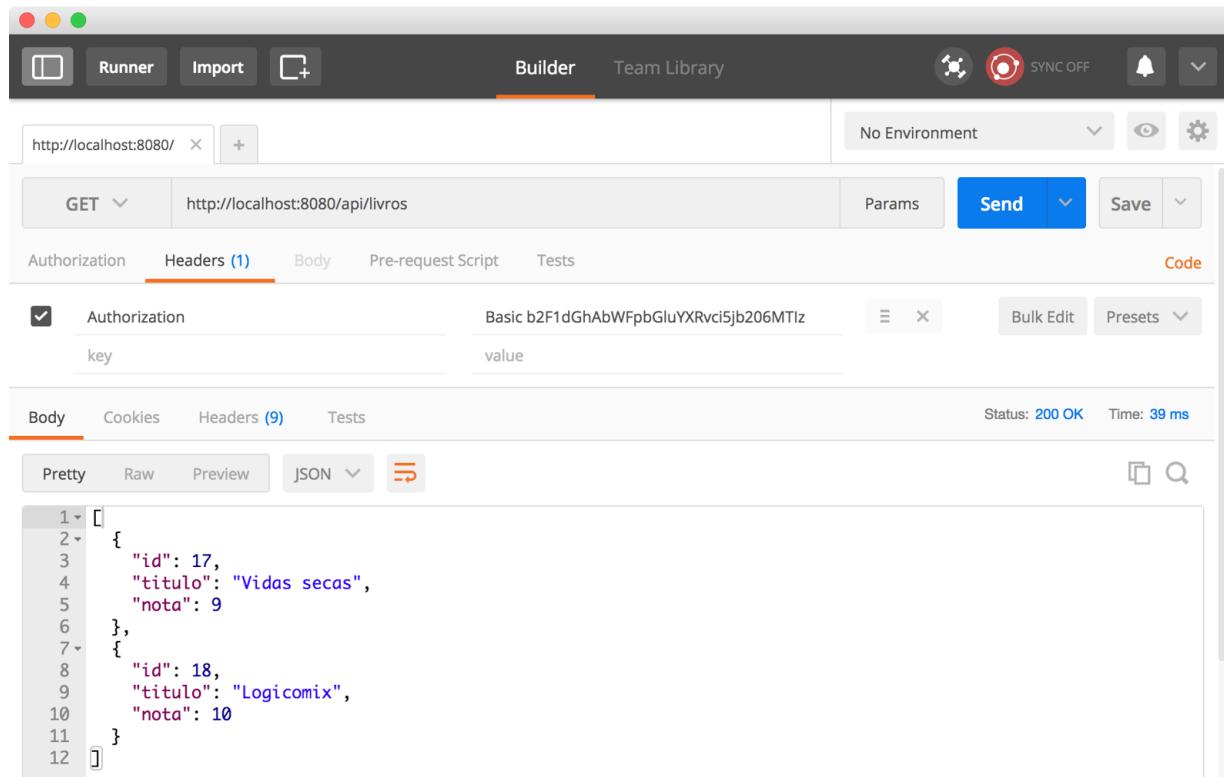
Com a configuração atual, o que acontece se você tentar enviar uma solicitação para o endpoint de consulta de livros? Caso estiver no Linux ou no Mac OS, execute o comando a seguir para consultar os livros de um determinado usuário informando as credenciais que você cadastrou em seu ambiente. No meu caso, usei o login `oauth@mailinator.com` e a senha `123`.

```
curl -X GET --user oauth@mailinator.com:123
"http://localhost:8080/api/livros"
```

O comando apresentado é um utilitário que permite enviar requisições `http` para um determinado endpoint, podendo para isso informar parâmetros de cabeçalho, URL, conteúdo de formulário e muito mais. Ao executar o comando apresentado, você deve obter na saída do terminal um conteúdo semelhante ao mostrado a seguir.

```
[{
    "id": 17,
    "titulo": "Vidas secas",
    "nota": 9
}]
```

Caso você não tenha o `cURL` instalado em sua máquina e queira outra opção, também é possível utilizar uma ferramenta chamada `Postman` que é distribuída como uma aplicação do `chrome`. Veja na figura a seguir como seria a mesma requisição de consulta de livros usando o `Postman`.



The screenshot shows the Postman application window. At the top, there are tabs for 'Runner', 'Import', and 'Builder' (which is selected). Below the tabs, there's a URL input field containing 'http://localhost:8080/' and a dropdown menu. To the right of the URL field, it says 'No Environment'. On the far right of the header bar are icons for sync, notifications, and settings.

In the main area, there's a 'GET' dropdown set to 'http://localhost:8080/api/livros'. Below this, there are tabs for 'Params', 'Send' (which is highlighted in blue), and 'Save'. Underneath these tabs, there are sections for 'Authorization', 'Headers (1)', 'Body', 'Pre-request Script', and 'Tests'. The 'Headers (1)' tab is currently active, showing a single entry: 'Authorization' with 'key' and 'value' fields both containing 'Basic b2F1dGhAbWFpbGluYXRvcj5jb206MTIz'. There are buttons for 'Bulk Edit' and 'Presets' to the right of this section.

Below the headers, the 'Body' tab is active. It has sub-tabs for 'Pretty', 'Raw', 'Preview', and 'JSON' (which is selected). The JSON preview area shows the following data:

```
1 [ { 2   "id": 17, 3   "titulo": "Vidas secas", 4   "nota": 9 5 }, 6   { 7     "id": 18, 8     "titulo": "Logicomix", 9     "nota": 10 10 } 11 ] 12 ]
```

Figura 4.4: Postman para acessar o endpoint de consulta de livros

Uma diferença aqui é que estamos passando o login e a senha do usuário de um jeito um pouco diferente do que enviamos através do comando `cURL`. No exemplo utilizando o `Postman`, estamos enviando as credenciais do usuário da seguinte maneira:

```
Basic b2F1dGhAbWFpbGluYXRvcj5jb206MTIz
```

Mas o que significa essa string cheia de caracteres sem sentido? Esse é o resultado obtido após montar a string `oauth@mailinator.com:123` codificada com base 64.

BASE 64

Atenção, pois ao contrário do que muitos pensam, Base 64 não é um mecanismo de criptografia. Base 64 é apenas um algoritmo para codificar dados que possam ser transmitidos via `HTTP`. Com esse algoritmo, é possível converter um conteúdo qualquer em um texto representado por uma quantidade limitada de caracteres (nesse caso, 64 caracteres).

Através do comando `CURL`, usamos o parâmetro `--user` para poder enviar as credenciais sem realizar a codificação manualmente.

Nesse caso, o `CURL` já faz esse trabalho para nós. Com o `Postman`, também é possível enviar as credenciais sem ter de se preocupar em codificar manualmente. Porém, o objetivo aqui não é entrar em detalhes sobre essas ferramentas. Caso estiver curioso, fique à vontade para pesquisar mais sobre o comando `CURL` ou o `Postman`.

Agora você pode estar se perguntando o seguinte: como é que, com a configuração que fizemos, o Spring Security sabe validar se as credenciais do usuário estão corretas? Não fizemos nenhuma configuração para isso, então como o usuário foi identificado e os seus livros puderam ser encontrados? Apesar de você não ter escrito código para isso, já existe no projeto uma classe responsável por recuperar um usuário do banco de dados através de seu login.

```
@Service
public class DadosDoUsuarioService implements UserDetailsService {

    @Autowired
    private Usuarios usuarios;

    @Override
    public UserDetails loadUserByUsername(String email)
        throws UsernameNotFoundException {
        Optional<Usuario> usuario = usuarios.buscarPorEmail(email);
```

```

        if (usuario.isPresent()) {
            return new ResourceOwner(usuario.get());
        } else {
            throw new UsernameNotFoundException("usuario não autorizado");
        }
    }

}

```

A classe `DadosDoUsuarioService` é a responsável por consultar um usuário através do login. Essa classe implementa uma interface especial do Spring Security chamada `UserDetailsService` e, além disso, está anotada com `@Service` (o que a torna reconhecida no contexto do Spring como um `bean`). Apesar de essa classe já ser carregada como um `bean` no contexto do Spring, como o Spring Security obtém acesso a esse objeto para realizar autenticação?

Ao habilitar o uso do Spring Security, um `AuthenticationManager` global é criado. Em seguida, o Spring Security procura por um `bean` que implemente `UserDetailsService` e injeta-o no `AuthenticationManager` global. O `bean` encontrado no nosso caso é justamente o que definimos através da classe `DadosDoUsuarioService`.

Portanto, a classe que valida as credenciais de um usuário já está sendo adicionada na configuração automaticamente para nós. Caso tiver a curiosidade de saber como essa configuração automática é realizada, dê uma olhada no código-fonte da classe `InitializeUserDetailsBeanManagerConfigurer` do Spring Security.

Mais uma questão a ser levantada é como a classe `DadosDoUsuarioService` obtém os dados do usuário que estão no banco de dados? Para isso, o método `loadUserByUsername` realiza uma chamada para o método `buscarPorEmail(email)` da classe `Usuarios`. A classe `Usuarios` que está anotada com `@Repository` é a que tem a responsabilidade de efetivamente fazer a busca dos dados do usuário no banco de dados.

```

@Repository
@Transactional
public class Usuarios {

    @PersistenceContext
    private EntityManager em;

    public Optional<Usuario> buscarPorEmail(String email) {
        TypedQuery<Usuario> query = em.createQuery(
            "select u from Usuario u where u.credenciais.email = :email",
            Usuario.class);

        query.setParameter("email", email);

        try {
            Usuario usuario = query.getSingleResult();
            return Optional.of(usuario);
        } catch (NoResultException e) {
            return Optional.empty();
        }
    }

    // outros métodos omitidos
}

```

Voltando para a classe `DadosDoUsuarioService`, vamos dar uma olhada na declaração do método que busca os dados do usuário através do login.

```
public UserDetails loadUserByUsername(String email);
```

Esse método retorna um objeto do tipo `UserDetails` que também não faz parte da regra de negócio. `UserDetails` é uma interface que representa um usuário autenticado no sistema. Um objeto do tipo `UserDetails` permite que sejam obtidas informações sobre o usuário, como credenciais, nome, permissões ou se ele está ativo na aplicação. Para satisfazer o contrato da interface `UserDetailsService`, já deixei pronta na aplicação `bookserver` uma classe chamada `ResourceOwner`.

```
public class ResourceOwner implements UserDetails {  
    private Usuario usuario;  
  
    public ResourceOwner(Usuario usuario) {  
        this.usuario = usuario;  
    }  
  
    // métodos da interface UserDetails omitidos  
}
```

Para obter o `username` e as credenciais do usuário, você é obrigado a implementar os métodos `getPassword` e `getUsername` conforme realizado no projeto de exemplo.

```
@Override  
public String getPassword() {  
    return usuario.getCredenciais().getSenha();  
}  
  
@Override  
public String getUsername() {  
    return usuario.getCredenciais().getEmail();  
}
```

Para facilitar a identificação do usuário logado, adicionei também um método customizado `getId`.

```
public Long getId() {  
    return usuario.getId();  
}
```

Apenas a título de curiosidade, caso você tenha outros objetos que implementem `UserDetailsService` e estes não sejam gerenciados pelo contexto do Spring, você também pode realizar a configuração manualmente sobrescrevendo o seguinte método `configure`:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {  
    auth.userDetailsService(new DadosDoUsuarioService());  
}
```

4.3 Recuperar os dados do Resource Owner

Quando criamos o `controller` para a API da aplicação `bookserver`, definimos o método `livros` que é responsável por recuperar a lista de livros de um determinado usuário. Seguindo o modelo de classes que foi definido, para obter os livros de uma estante do usuário logado, usamos o seguinte trecho de código:

```
Estante estante = donoDosLivros().getEstante();
```

Mas como foi que fizemos para recuperar os livros do usuário que foi autenticado durante a requisição? O que o método `donoDosLivros` faz para recuperar o usuário atual?

```
private Usuario donoDosLivros() {
    Authentication authentication = SecurityContextHolder
        .getContext().getAuthentication();

    ResourceOwner donoDosLivros =
        (ResourceOwner) authentication.getPrincipal();

    return usuarios.buscarPorID(donoDosLivros.getId());
}
```

Durante a fase de autenticação, caso o usuário exista no banco de dados e tenha passado as credenciais corretas, ele é armazenado no contexto da requisição dentro de um objeto do tipo `Authentication`.

Através desse objeto, podemos obter as credenciais, permissões, detalhes do usuário e o objeto que representa o usuário logado no sistema. No nosso caso, o objeto que representa o usuário logado é definido através da classe `ResourceOwner` que, por sua vez, implementa `UserDetails`.

A classe `ResourceOwner` possui um método que nos permite recuperar o `id` do usuário. Dessa forma, podemos usar o repositório de usuários para buscar todos os detalhes do usuário que estão armazenados no banco de dados e, inclusive, a sua lista de livros.

Repare que, apesar de ainda não estarmos usando OAuth 2.0, já estamos usando o nome `ResourceOwner` para representar o usuário que é dono dos recursos na aplicação `bookserver`. Assim já temos bem claro que, ao acessar a API de consulta de livros, estamos acessando recursos de um usuário.

4.4 Conclusão

Ao estudar este capítulo, você já tem familiaridade suficiente com o framework Spring Security para começar a usar o Spring Security OAuth2 com tranquilidade. Para usar o suporte a OAuth 2.0, fornecido pelo Spring, a documentação disponível no site do framework pode ser suficiente para iniciar um projeto.

Porém, à medida que as customizações são necessárias e que bugs complicados começam a aparecer no sistema, conhecer a fundo o framework que está sendo utilizado pode ser de grande ajuda. Como será visto nos capítulos a seguir, muito do que foi visto até aqui será base para configurar um projeto usando OAuth 2.0.

CAPÍTULO 5

Exemplo de aplicação cliente

Com a aplicação `bookserver` funcionando e com uma API pronta para ser acessada, já podemos permitir a integração com a primeira aplicação terceira. Mas como uma aplicação terceira pode se integrar para usar nossa API? Como esse tipo de **Client** pode ser implementado tendo de utilizar as credenciais do usuário para recuperar seus recursos cadastrados na aplicação `bookserver` ?

Para responder a essas perguntas, criei uma aplicação `client` para tal propósito. Trata-se de uma aplicação web que simula uma rede social de profissionais em que um usuário pode permitir a exibição dos livros que já leu em sua linha do tempo. E como os livros que o usuário já leu estão disponíveis na aplicação `bookserver`, nada mais justo do que realizar uma integração acessando esses dados via API.

Por meio dessa aplicação, quero mostrar na prática como a integração pode ser feita e quais os problemas que vamos encontrar ao usar a abordagem de repassar as credenciais durante o acesso aos recursos do usuário. Durante os próximos capítulos, também evoluiremos passo a passo essa aplicação para utilizar OAuth 2.0.

Para acompanhar este capítulo, importe para sua IDE os projetos `client` e `bookserver` disponíveis no diretório `livro-spring-oauth2/capitulo-5`. O projeto `client` também utiliza Maven, assim como o projeto `bookserver`.

O projeto `client` usa o Spring Boot para facilitar o desenvolvimento da aplicação web e possui praticamente as mesmas dependências do projeto `bookserver`. Portanto, não entrarei em detalhes sobre as dependências para as bibliotecas e frameworks utilizados. O importante agora é se familiarizar com essa aplicação através da navegação nas telas do sistema e explorar o funcionamento das

```
classes BookserverService , MinhaContaController e  
IntegracaoController .
```

Entretanto, para que você possa executar a aplicação, é necessário criar o banco de dados no MySQL executando os comandos disponíveis no script `database.sql`. Este se encontra na raiz do projeto `client`.

```
CREATE DATABASE clientapp;  
CREATE USER 'clientapp'@'localhost' IDENTIFIED BY '123';  
GRANT ALL PRIVILEGES ON clientapp.* TO 'clientapp'@'localhost';  
  
use clientapp;  
  
create table usuario(  
    id int auto_increment primary key,  
    nome varchar(100),  
    login varchar(100),  
    senha varchar(50),  
    login_bookserver varchar(100) NULL,  
    senha_bookserver varchar(50) NULL  
);
```

Temos apenas uma tabela no banco de dados `clientapp` que é a `usuario`. A ideia aqui é mostrar como podem ser armazenadas as credenciais do usuário na aplicação `bookserver` e como isso pode ser perigoso.

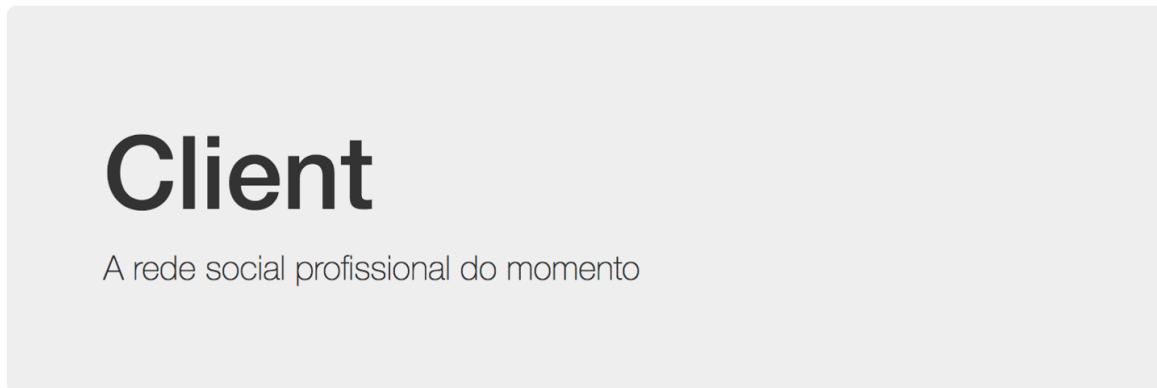
5.1 O funcionamento da aplicação client

Com o banco de dados criado, já é possível iniciar a aplicação. Para isso, execute através da sua IDE a classe `BookserverClientApplication`, ou execute no terminal o comando a seguir.

Não esqueça que a aplicação bookserver também precisa ser iniciada!

```
mvn spring-boot:run
```

Assim que a aplicação iniciar, acesse a URL <http://localhost:9000> para acessar a `home` da aplicação `Client`.



Login

Faça o login na aplicação para poder acessar sua rede social ou cadastre-se. Ao se autenticar você pode publicar conteúdos em sua linha do tempo.

[Login](#)

Acesse sua conta

Ao acessar a conta você pode ver os conteúdos que publicou. É aqui que você também pode mostrar os livros que você já cadastrou na aplicação bookserver

[Minha conta](#)

Figura 5.1: Tela inicial do Client

Tente clicar em minha conta e você será redirecionado para a tela de login. Mas como ainda não temos um cadastro nessa aplicação, clique no link `Faça seu cadastro`, ou acesse diretamente a URL <http://localhost:9000/usuarios>.

Client

Cadastre-se

Nome:

Login:

Senha:

Enviar

Cancelar

Figura 5.2: Tela de cadastro

Aqui você pode cadastrar um usuário qualquer pois, para acessar os recursos do usuário, não vamos utilizar as mesmas credenciais. Assim que fizer o cadastro, o usuário será autenticado no sistema e já será redirecionado para a tela principal na qual ele pode ver a própria linha do tempo.

Client

Olá adolfo@email.com

Atividades recentes

dia	atividade
hoje	curtiu um post sobre segurança
ontem	participou de uma maratona de programação
semana passada	participou de um Dojo de node.js em Köln Mülheim

 Você tem conta no bookserver?
Mostre [aqui](#) os livros que você já leu para melhorar seu perfil.

[Voltar](#)



Figura 5.3: Tela principal da conta do usuário

Nessa tela, você pode ver alguns registros fixos na timeline do usuário e um aviso informando que o usuário pode mostrar também os livros que possui cadastrados na aplicação `bookserver`. Clique no link para mostrar os livros, ou acesse a URL <http://localhost:9000/integracao>.

Client

adolfo@email.com, faça aqui a integração com o bookserver

Forneça as credenciais do bookserver para que possamos recuperar seus livros

Login:

oauth@mailinator.com

Senha:

...

Enviar

Home

Figura 5.4: Tela de integração com bookserver

Nessa tela, o usuário pode informar o login e a senha que utiliza para acessar o sistema bookserver . Dessa forma, o Client guardará essas credenciais para poder recuperar os livros do usuário e mostrar no seu perfil configurado na rede social fictícia. No meu caso, utilizei um usuário que já cadastrei no bookserver cujo login é oauth@mailinator.com e a senha é 123 . Ao confirmar os dados, os livros da aplicação bookserver já vão aparecer na tela principal do usuário.

Olá adolfo@email.com

Atividades recentes

dia	atividade
hoje	curtiu um post sobre segurança
ontem	participou de uma maratona de programação
semana passada	participou de um Dojo de node.js em Köln Mülheim

Esses são seus livros cadastrados na aplicação bookserver

titulo	nota
Vidas secas	9
Logicomix	10

Figura 5.5: Tela principal da conta do usuário após a integração

5.2 Integração repassando as credenciais

Agora como tudo isso é feito na aplicação? Como as credenciais são armazenadas e como elas são usadas para recuperar os livros do usuário? Quando o usuário autoriza o acesso aos seus recursos fornecendo seu login e senha registrados na aplicação `bookserver`, esses dados são persistidos no banco de dados para uso posterior. Esse processo é realizado dentro do método `autorizar` da classe `IntegracaoController`.

```
@RequestMapping(method = RequestMethod.POST)
public ModelAndView autorizar(Autorizacao autorizacao) {

    AcessoBookserver acessoBookserver = new AcessoBookserver();
    acessoBookserver.setLogin(autorizacao.getLogin());
    acessoBookserver.setSenha(autorizacao.getSenha());
```

```

        Usuario usuario = usuarioLogado();
        usuario.setAcessoBookserver(acessoBookserver);

        usuarios.save(usuario);

        return new ModelAndView("redirect:/minhaconta/principal");
    }
}

```

Do jeito que está sendo feita a integração entre o `client` e a aplicação `bookserver`, as credenciais do usuário ficam duplicadas em dois sistemas, conforme podemos ver a seguir.

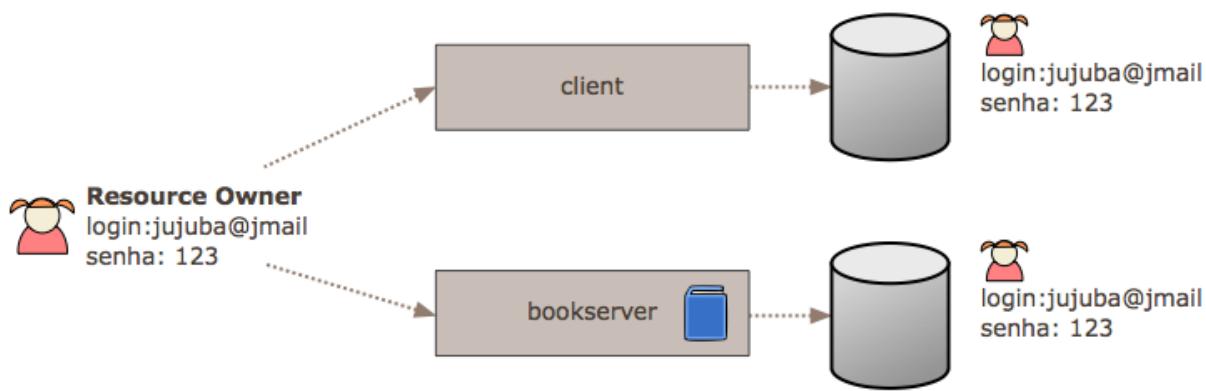


Figura 5.6: Credenciais do usuário em dois sistemas diferentes

Além da duplicidade de dados, a aplicação está armazenando as credenciais sem utilizar nenhum mecanismo de criptografia. Até poderíamos usar criptografia para persistir os dados, porém, ainda assim, como o usuário pode confiar que o `client` não vai utilizar seus dados para outros propósitos?

Para a sorte dos usuários, o `client` está usando essas credenciais apenas para consultar os dados do cliente, conforme mostrado no código-fonte do método `livros` da classe `BookserverService`:

```

public List<Livro> livros(BasicAuthentication credenciais) throws
UsuarioSemAutorizacaoException {
    RestTemplate restTemplate = new RestTemplate();

    MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
}

```

```

        headers.add("Authorization", "Basic " +
        credenciais.getCredenciaisBase64()));

        String endpoint = "http://localhost:8080/api/livros";

        RequestEntity<Object> request = new RequestEntity<Object>(
            headers, HttpMethod.GET, URI.create(endpoint)
        );

        try {
            ResponseEntity<Livro[]> resposta = restTemplate.exchange(request,
            Livro[].class);
            if (resposta.getStatusCode().is2xxSuccessful()) {
                return listaFromArray(resposta.getBody());
            } else {
                throw new RuntimeException("sem sucesso");
            }
        } catch (HttpClientErrorException e) {
            throw new UsuarioSemAutorizacaoException("não foi possível obter
            os livros do usuário");
        }

    }
}

```

Vamos entender o que está acontecendo nesse código que parece ser complicado. Para poder realizar a requisição `http` no endpoint de consulta de livros, estamos usando nesse código uma instância de `RestTemplate`. Essa classe do Spring Web permite montar uma requisição `http` para poder acessar um serviço REST.

Repare que, para montar a requisição, estamos informando o cabeçalho `http Authorization` e qual é o endpoint de consulta de livros. Ao obter a resposta através da chamada

`restTemplate.exchange(request, Livro[].class);`, transformamos o que foi retornado para uma lista de livros utilizando um método privado `listaFromArray`. Caso ocorra algum erro durante a requisição ou o status `http` seja diferente de `2xx`, lançamos uma exceção. Qualquer status `http` que comece com `2` representa sucesso.

Observe também que, ao definir o cabeçalho `Authorization`, estamos criando a `String` com as credenciais assim como mostrei anteriormente utilizando o `Postman`. Ou seja, estamos codificando as credenciais com `base64` da seguinte maneira:

```
@AllArgsConstructor
public class BasicAuthentication {

    @Getter
    private String login;

    @Getter
    private String senha;

    public String getCredenciaisBase64() {
        String credenciais = login + ":" + senha;
        String credenciaisCodificadasComBase64 = new String(
            Base64.getEncoder().encode(credenciais.getBytes()));
        return credenciaisCodificadasComBase64;
    }
}
```

O `service` definido na classe `BookserverService` é usado pelo `controller` `MinhaContaController` para obter os livros do usuário logado. Veja como ficou o método que está mapeado para a URL `/minhaconta/principal`.

```
@RequestMapping(value = "/principal")
public ModelAndView principal() {

    Usuario usuario = usuarioLogado();
    String login = usuario.getAcessoBookserver().getLogin();
    String senha = usuario.getAcessoBookserver().getSenha();

    ModelAndView mv = new ModelAndView("minhaconta/principal");

    BasicAuthentication credenciais =
        new BasicAuthentication(login, senha);

    try {
```

```
        mv.addObject("livros", bookserverService.livros(credenciais));
    } catch (UsuarioSemAutorizacaoException e) {
        mv.addObject("erro", e.getMessage());
    }

    return mv;
}
```

5.3 Conclusão

Por meio deste curto capítulo, apresentei a aplicação `client` que melhoraremos ao decorrer do livro, utilizando o protocolo OAuth 2.0. É importante que você conheça tanto o lado servidor quanto o `client` nessa integração realizada através de APIs. Dessa forma, poderá entender o porquê de algumas soluções serem mais frágeis do que outras, e como tais fragilidades podem comprometer a segurança do sistema e dos dados dos usuários.

Agora que já temos um exemplo de aplicação `client` que repassa as credenciais do usuário para acessar seus recursos, vou mostrar como melhorar a segurança e a confiabilidade das aplicações usando o protocolo OAuth 2.0 com o Spring Security OAuth2.

CAPÍTULO 6

O Spring Security OAuth2

Agora é hora de começar a entender mais a fundo o protocolo OAuth 2.0 e como utilizar o Spring Security OAuth2 para proteger a API fornecida pela aplicação `bookserver`. Antes de tudo, é preciso definir algumas questões arquiteturais da aplicação `bookserver` e decidir qual fluxo do OAuth 2.0 será usado para proteger a aplicação.

Através deste capítulo, você verá como adicionar suporte ao Spring Security OAuth2 para configurar um **OAuth Provider**, que pode ser implementado como uma única aplicação ou separado em dois sistemas diferentes.

6.1 Authorization Server e Resource Server juntos

A figura a seguir apresenta uma visão de alto nível de como deve ficar a aplicação `bookserver` quando o suporte ao OAuth 2.0 for adicionado. De acordo com a imagem, a camada de segurança continuará utilizando *HTTP Basic Authentication* em conjunto com o OAuth 2.0 para proteger a API.

Como podemos observar, o OAuth Provider está sendo representado como uma única aplicação que mantém o Resource Server e o Authorization Server juntos. Vamos usar essa arquitetura inicialmente apenas por questões de facilidade.

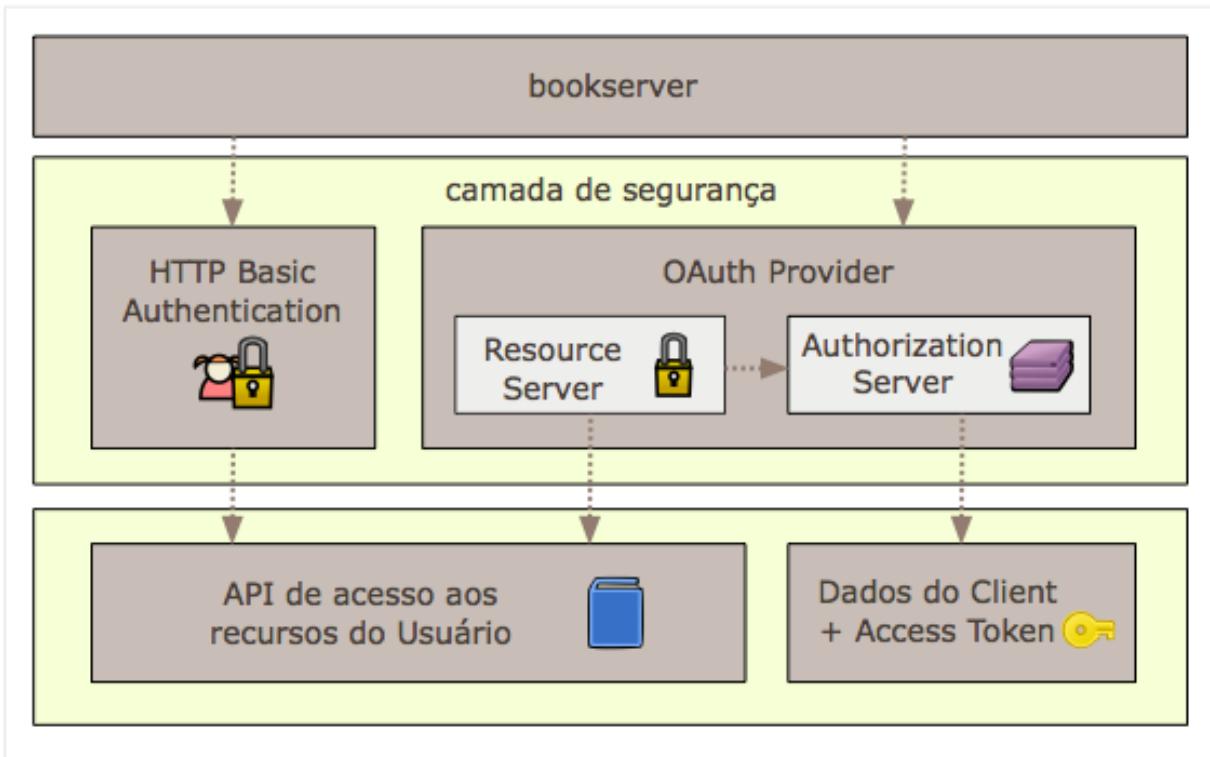


Figura 6.1: Arquitetura OAuth 2.0 do projeto bookserver

Essa abordagem possui algumas fragilidades conforme explicarei mais adiante. Não se preocupe por enquanto, pois a ideia agora é que você entenda como configurar o OAuth 2.0 com o Spring, e como funcionam os fluxos principais para autorização e obtenção de tokens de acesso.

6.2 Authorization Server e Resource Server separados

Algumas vezes é interessante configurar o Authorization Server e o Resource Server como aplicações separadas. Implementar o OAuth Provider de forma separada favorece a segurança do sistema como um todo se considerarmos o princípio da **redução da superfície de ataque**. Esse princípio parte da ideia de que os riscos envolvendo a segurança de uma aplicação aumentam a cada nova funcionalidade

adicionada. Separando as aplicações, teremos menos funcionalidades em cada uma delas, pois estamos dividindo bem as responsabilidades.

Outro motivo para implementar um OAuth Provider de forma separada é quando temos um ambiente no qual diversas aplicações interagem entre si, consumindo recursos em benefício próprio ou mesmo em benefício de algum usuário. Esse tipo de cenário é bem conhecido atualmente quando se pensa em uma arquitetura baseada em *microservices*.

Em poucas palavras, o termo em inglês *microservices* (microsserviços), de acordo com o livro *Building Microservices* de Sam Newman, significa serviços pequenos e autônomos que trabalham em conjunto onde cada um tem responsabilidades bem definidas.

Quando trabalhamos com *microservices*, é possível se deparar com uma arquitetura em que cada pequeno serviço precisa se autenticar para acessar recursos uns dos outros. Cada serviço pode ser tratado como um Resource Server protegido por OAuth 2.0.

Imagine se tivéssemos de implementar um Authorization Server para cada Resource Server. Em um cenário assim, faz bastante sentido centralizar o gerenciamento de tokens de acesso e registro dos Clients em um único Authorization Server, conforme mostrado na figura seguir.

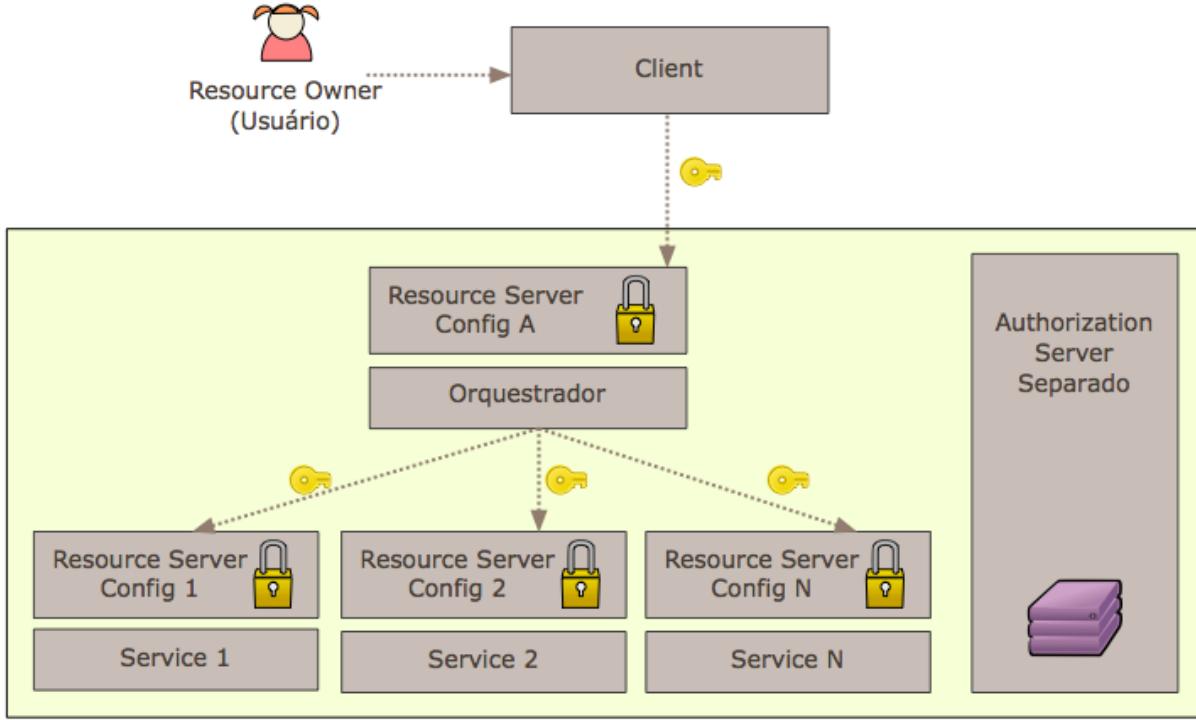


Figura 6.2: Authorization Server e Resource Server separados

A figura anterior mostra uma arquitetura um pouco mais complexa, na qual temos um Client que acessa recursos em nome de um Resource Owner, assim como temos aplicações que acessam recursos em benefício próprio. Em um cenário como o apresentado, pode ser necessário utilizar *grant types* diferentes para cada Client. Considerando o Client externo que acessa recursos do usuário, poderíamos usar o grant type Authorization Code, Implicit ou até mesmo o Resource Owner Password Credentials.

Para as aplicações dentro do domínio do Server, o ideal seria usar o grant type Client Credentials, pois as aplicações vão acessar recursos em benefício próprio. A validação dos tokens de acesso recebidos por cada Resource Server é realizada por apenas um Authorization Server.

Esse tipo de arquitetura também deve ser usado com cuidado, pois caso muitos Resource Servers precisem validar tokens de acesso em um curto intervalo de tempo, com muita frequência, o

Authorization Server pode não suportar o grande número de requisições. Para resolver esse tipo de problema, podemos usar tokens de acesso autocontidos, que serão explicados mais adiante quando adicionarmos suporte ao JWT (*JSON Web Token*) na aplicação `bookserver`.

6.3 Configuração do Spring Security OAuth2

Do mesmo modo que foi feita a configuração da segurança com *HTTP Basic Authentication*, é preciso seguir alguns passos similares para configurar o OAuth 2.0. Agora, vamos melhorar a segurança do projeto `bookserver` utilizando OAuth 2.0.

Para isso, **importe** o projeto `bookserver` que se encontra no diretório `livro-spring-oauth2/capitulo-6`. O primeiro passo a ser seguido é adicionar a dependência do Spring Security OAuth2 no arquivo `pom.xml`.

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

PROBLEMA COM VERSÕES MAIS ANTIGAS DO SPRING SECURITY OAuth2

Um ponto importante a ser mencionado é que, quando você estiver trabalhando em um projeto com o Resource Server e o Authorization Server na **mesma aplicação**, é necessário usar versões superiores a 2.0.8 do Spring Security OAuth2. Um bug que causava problemas com os endpoints de autorização do OAuth 2.0 foi apontado no GitHub (e logo corrigido), conforme pode ser visto em <https://github.com/spring-projects/spring-security-oauth/issues/634>.

Como estamos usando a versão 1.4.3 do Spring Boot no projeto bookserver , o esquema de autoconfiguração existente no próprio Spring Boot já importa a versão 2.0.12 do Spring Security OAuth2 para nós.

Com a dependência do Spring Security OAuth2 adicionada, o próximo passo é criar a classe de configuração utilizando a anotação `@Configuration`. Crie para isso a classe `ConfiguracaoOAuth2` no pacote `br.com.casadocodigo.seguranca.oauth` do projeto `bookserver` . A ideia aqui é que essa classe possa agrupar as configurações do Resource Server e do Authorization Server através de classes internas.

A constante `RESOURCE_ID` mostrada no exemplo serve para identificar o nome do Resource Server que estamos criando. Esse nome é usado pelo Authorization Server para definir a **audiência** do token, ou seja, para qual Resource Server um token foi gerado.

```
@Configuration  
public class ConfiguracaoOAuth2 {  
    public static final String RESOURCE_ID = "books";  
}
```

6.4 Configuração do Resource Server

Vamos então criar a configuração do Resource Server através de uma classe interna chamada `OAuth2ResourceServer`, de acordo com o código seguinte:

```
@Configuration
public class ConfiguracaoOAuth2 {

    public static final String RESOURCE_ID = "books";

    @EnableResourceServer
    protected static class OAuth2ResourceServer extends
        ResourceServerConfigurerAdapter {

        @Override
        public void configure(ResourceServerSecurityConfigurer resources)
            throws Exception {
            resources.resourceId(RESOURCE_ID);
        }

    }
}
```

O código anterior possui apenas um método de configuração que identifica o Resource Server que estamos configurando. Apesar de ainda não haver nenhuma lógica escrita no código da classe `OAuth2ResourceServer`, o fato de anotar a classe com `@EnableResourceServer` já faz com que bastante coisa aconteça durante a inicialização da aplicação. Ao adicionar a anotação `@EnableResourceServer`, uma configuração `default` é importada através da classe `ResourceServerConfiguration`, conforme pode ser observado no código-fonte da anotação.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(ResourceServerConfiguration.class)
```

```
public @interface EnableResourceServer {  
}
```

A classe `ResourceServerConfiguration` possui uma semelhança muito interessante com a classe `ConfiguracaoDeSeguranca`. Assim como a `ConfiguracaoDeSeguranca`, a classe `ResourceServerConfiguration` também estende `WebSecurityConfigurerAdapter` e sobrescreve o método `configure(HttpSecurity http)`. Repare como o próprio Spring Security OAuth2 utiliza o Spring Security para dar suporte ao protocolo OAuth 2.0 de um jeito parecido com o que fizemos no capítulo anterior.

```
@Configuration  
public class ResourceServerConfiguration  
    extends WebSecurityConfigurerAdapter implements Ordered {  
  
    // atributos e outros métodos omitidos  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        // implementação omitida  
    }  
}
```

O Spring Security OAuth2 utiliza os **builders** que expliquei no capítulo passado para criar a camada de segurança do projeto através de filtros encadeados pela classe `FilterChainProxy`. Apenas para ajudar você a refrescar a memória, os **builders** de que estou falando são o `WebSecurity` e o `HttpSecurity`, que podem ser customizados através de **configurers**.

A classe `ResourceServerConfiguration`, além de ser de configuração, também se comporta como um **configurer**, pois herda de `WebSecurityConfigurerAdapter` que, por sua vez, implementa indiretamente a interface `SecurityConfigurer`. Porém, onde a lógica do OAuth 2.0 é inserida? Quem é responsável por interceptar as requisições que chegam para os endpoints protegidos por OAuth 2.0?

A lógica de proteção via OAuth 2.0 é realizada pelo filtro `OAuth2AuthenticationProcessingFilter`, adicionado na cadeia de filtros criada pelo **builder** `HttpSecurity`. Para que esse filtro possa ser embutido dentro da cadeia de filtros do `FilterChainProxy`, a classe `ResourceServerConfiguration` utiliza um outro objeto do tipo `ResourceServerSecurityConfigurer`. Este é criado dentro do método `configure(HttpSecurity http)` da classe `ResourceServerConfiguration`, e é passado para o **builder** `HttpSecurity` conforme mostrado a seguir:

```
@Configuration
public class ResourceServerConfiguration
    extends WebSecurityConfigurerAdapter implements Ordered {
    // atributos omitidos
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        ResourceServerSecurityConfigurer resources = new
        ResourceServerSecurityConfigurer();
        // códigos omitidos
        http.apply(resources);
        // códigos omitidos
    }
    // outros métodos omitidos
}
```

Durante a inicialização da aplicação, quando o **builder** `HttpSecurity` invocar a callback de configuração

`ResourceServerSecurityConfigurer.configure(http)`, o filtro do OAuth 2.0 será criado, configurado e adicionado na cadeia de filtros do Spring Security, conforme mostrado a seguir:

```
public final class ResourceServerSecurityConfigurer extends
    SecurityConfigurerAdapter<DefaultSecurityFilterChain,
    HttpSecurity> {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        AuthenticationManager oauthAuthenticationManager =
        oauthAuthenticationManager(http);
        resourcesServerFilter = new
```

```

OAuth2AuthenticationProcessingFilter();

    // configurações do resourceServerFilter omitidas

    // @formatter:off
    http
        .authorizeRequests().expressionHandler(expressionHandler)
        .and()
        .addFilterBefore(resourcesServerFilter,
AbstractPreAuthenticatedProcessingFilter.class)
        .exceptionHandling()
        .accessDeniedHandler(accessDeniedHandler)
        .authenticationEntryPoint(authenticationEntryPoint);
    // @formatter:on
}
}

```

Com todo esse trabalho sendo realizado, o único ponto que nós, desenvolvedores, precisamos nos preocupar é com configurações. Não precisamos implementar detalhes do protocolo OAuth 2.0.

Para começar a proteger os endpoints com OAuth 2.0, basta definirmos quais URLs serão interceptadas pelo filtro do Spring Security OAuth2. Para realizar essa configuração na classe `OAuth2ResourceServer` que acabamos de criar, vamos implementar o método de callback `configure(HttpSecurity http)` conforme mostrado a seguir:

```

@EnableResourceServer
protected static class OAuth2ResourceServer extends
ResourceServerConfigurerAdapter {

    @Override
    public void configure(ResourceServerSecurityConfigurer resources)
        throws Exception {
        resources.resourceId(RESOURCE_ID);
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {

```

```
    http
        .authorizeRequests()
            .anyRequest().authenticated().and()
        .requestMatchers()
            .antMatchers("/api/v2/livros");
    }
}
```

No código anterior, estamos dizendo para o filtro proteger o endpoint `/api/v2/livros`, porém ainda não temos esse endpoint mapeado em nenhum controller da aplicação. Adicione esse mapeamento na anotação `@RequestMapping` da classe `LivrosApiController`.

```
@Controller
@RequestMapping({"/api/livros", "/api/v2/livros"})
public class LivrosController {
    // código omitido
}
```

Do jeito que configuramos a aplicação, quando o endpoint `/api/livros` for acessado, deverá ser usado *HTTP Basic Authentication* como mecanismo de autenticação. Já para o endpoint `/api/v2/livros`, é necessário acessar utilizando um token OAuth.

Mas o que acontece se tentarmos consultar livros do usuário agora através do endpoint `/api/v2/livros`, repassando as credenciais do usuário através de *HTTP Basic Authentication*?

```
curl --user oauth@mailinator.com:123 "http://localhost:8080/api/v2/livros"
```

Receberemos como resposta um código de retorno HTTP `401 Unauthorized` e o conteúdo visto a seguir no corpo da resposta. Isso é ótimo, pois significa que a configuração que fizemos para o Resource Server já está protegendo a API através do protocolo OAuth 2.0.

```
{
    "error": "unauthorized",
    "error_description": "Full authentication is required to access this
```

```
resource"  
}
```

Para acessar o endpoint `/api/v2/livros`, precisamos enviar um token de acesso através do atributo `Authorization` do cabeçalho da requisição `http`. O problema é que ainda não temos esse token e quem é responsável por criar o token de acesso e solicitar as permissões do usuário é o **Authorization Server**.

6.5 Configuração inicial do Authorization Server

Precisamos então configurar o Authorization Server. Isso também será realizado através da declaração de uma classe interna dentro da classe `ConfiguracaoOAuth2`.

```
@Configuration  
public class ConfiguracaoOAuth2 {  
  
    @EnableAuthorizationServer  
    protected static class OAuth2AuthorizationServer  
        extends AuthorizationServerConfigurerAdapter {  
  
    }  
  
    @EnableResourceServer  
    protected static class OAuth2ResourceServer  
        extends ResourceServerConfigurerAdapter {  
            // código do resource server omitido  
        }  
}
```

A classe interna de configuração do Authorization Server foi criada com o nome `OAuth2AuthorizationServer` e está anotada com `@EnableAuthorizationServer`. Repare que, além da anotação que foi adicionada, a classe de configuração do Authorization Server estende a classe `AuthorizationServerConfigurerAdapter`.

A classe `AuthorizationServerConfigurerAdapter` implementa indiretamente a interface `SecurityConfigurer`. Dessa forma, assim como outros **configurers**, a classe

`AuthorizationServerConfigurerAdapter` fornece alguns métodos de callback, que podemos sobrescrever para configurar como os **builders** devem criar as camadas de segurança da aplicação.

Conforme já falado anteriormente, o Authorization Server possui as seguintes responsabilidades:

- Permitir que o usuário autorize o acesso aos seus recursos sem precisar passar suas credenciais para o Client;
- Permitir a geração do token de acesso através do fluxo de obtenção do token de acesso (grant type).

Portanto, para dar suporte a tais responsabilidades, o Authorization Server deve fornecer os endpoints para autorização e obtenção de tokens de acesso. Então como o Spring Security OAuth2 disponibiliza esses endpoints?

Ao adicionar a anotação `@EnableAuthorizationServer`, são importadas as classes de configuração `AuthorizationServerEndpointsConfiguration` e `AuthorizationServerSecurityConfiguration`. Podemos perceber de maneira bastante intuitiva que a classe

`AuthorizationServerEndpointsConfiguration` é responsável por configurar os endpoints do Authorization Server. Essa classe cria os seguintes beans que atuam como *controllers* dentro da aplicação:

- `AuthorizationEndpoint` : esse bean disponibiliza o endpoint `/oauth/authorize` e é usado para obter a autorização do usuário.
- `TokenEndpoint` : esse bean disponibiliza o endpoint `/oauth/token` para permitir que o Client faça a solicitação do token de acesso.

Agora temos definidos os *endpoints* que o Authorization Server deve fornecer segundo a especificação do protocolo OAuth 2.0. Porém, para que o Client possa utilizar os *endpoints* para obter autorização e solicitar tokens de acesso, ele precisa se **registrar** no Authorization Server.

6.6 Registro do Client no Authorization Server

Pois então, como fazer o registro do Client? A forma como o registro do Client é realizada não é definida pela especificação do protocolo OAuth 2.0. O importante é que o Client seja reconhecido pelo Authorization Server. Poderíamos em nosso caso criar uma tela de registro para o Client, em que ele informaria o nome da aplicação, URL de redirecionamento, logotipo da aplicação e muitas outras coisas que fossem necessárias.

No entanto, vamos utilizar uma solução mais simples para facilitar o entendimento do Spring Security OAuth2 e do protocolo em si. Para isso, configuraremos os dados do Client manualmente implementando o método `configure` mostrado no próximo trecho de código.

```
@EnableAuthorizationServer
protected static class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients
            .inMemory()
            .withClient("cliente-app")
            .secret("123456")
            .resourceIds(RESOURCE_ID);
    }

}
```

No método `configure`, estamos usando uma **configuração em memória** para manter os dados do Client, o que não é indicado para ser usado em produção. O Spring Security fornece maneiras bem melhores para mantermos esses dados (*Client details*), como por

exemplo, usar um banco de dados. Mais adiante, mostrarei como configurar o Authorization Server para usar o MySQL.

Na configuração manual que estamos fazendo, estamos definindo qual é o `client_id` e o `client_secret` do Client. Em um sistema real, geralmente esses dados são gerados automaticamente para o dono da aplicação que vai se integrar via OAuth 2.0. Além disso, o `client_id` e o `client_secret` devem ser únicos por Client. Também estamos referenciando o Resource Server relacionado à constante `RESOURCE_ID`, mas ainda não se preocupe com isso, pois voltaremos nesse assunto mais à frente.

Essa configuração de *client details* será usada para simularmos os acessos de um Client à aplicação `bookserver`. Entretanto, do jeito que o Authorization Server está configurado, não é possível acessar os endpoints protegidos por OAuth 2.0, pois é preciso definir qual fluxo do OAuth 2.0, ou seja, qual *grant type* será utilizado.

Anteriormente, apresentei alguns *grant types* e um fluxo básico para autorização e obtenção do token de acesso, que por sinal é o grant type **Authorization Code**. Apesar de esse ser um dos grant types mais conhecidos e usados, ele é um pouco mais complicado do que os outros grant types. Portanto, no próximo capítulo, vamos finalizar a configuração do Authorization Server utilizando um grant type mais simples, para que inicialmente você se habitue com o Spring Security OAuth2.

6.7 Conclusão

Neste capítulo, você já começou a ter uma noção sobre a configuração de um **OAuth Provider** utilizando o Spring Security OAuth2. Além disso, apresentei alguns modelos arquiteturais de distribuição da aplicação, considerando a implementação do **OAuth Provider** como módulos separados ou unificados. Agora você

também já entende como o Spring Security OAuth2 utiliza o Spring Security como pilar. Nos próximos capítulos, veremos mais detalhes sobre como e quando utilizar cada *grant type*, além de entender a fundo como cada participante do protocolo OAuth 2.0 interage entre si.

CAPÍTULO 7

Password credentials grant type

A partir deste capítulo, começarei a detalhar o grant type **Resource Owner Password Credentials**, pois esse é o fluxo que vamos configurar inicialmente na aplicação `bookserver`. A ideia é que, ao ler este capítulo, você fique familiarizado com a configuração de um Authorization Server usando o Spring Security OAuth2. Para cada grant type que for detalhado, você conhecerá como as mensagens devem ser trocadas entre as aplicações e como evitar problemas em cada situação.

Vamos entender um pouco do fluxo para obtenção de token de acesso usando o grant type Resource Owner Password Credentials, analisando a figura a seguir.

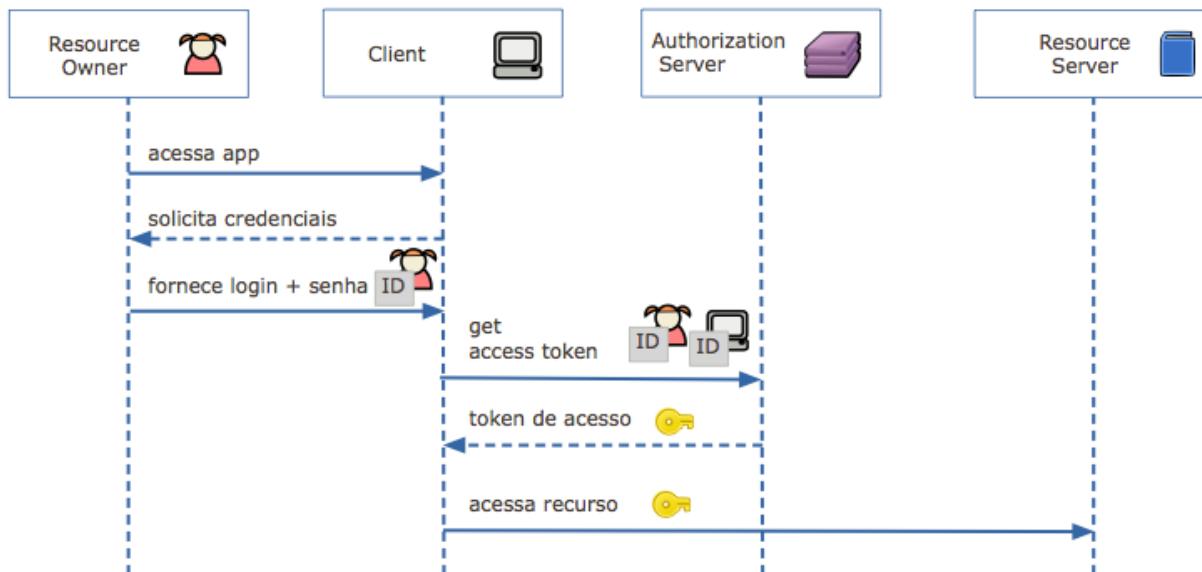


Figura 7.1: Resource Owner Password Credentials grant type

Repare que o fluxo apresentado na imagem já é bastante semelhante com o fluxo que repassa as credenciais do usuário. A principal diferença entre o grant type Resource Owner Password Credentials e a abordagem que já estamos usando até agora é que,

com OAuth 2.0, a aplicação Client solicita as credenciais do usuário apenas para obter um token junto ao Authorization Server. Por meio desse token obtido é que o Client pode acessar os endpoints protegidos pelo Resource Server, ou seja, através do fluxo OAuth 2.0, não precisamos mais repassar as credenciais do usuário.

Antes de começar a explorar o código, importe os projetos `bookserver` e `client` que estão no diretório `livro-spring-oauth2/capitulo-7`. Após importá-los, vamos analisar o código-fonte do `Client` atual para relembrar como estamos lidando com as credenciais do usuário nesse momento. Observe o trecho do método `autorizar` da classe `IntegracaoController`:

```
AcessoBookserver acessoBookserver = new AcessoBookserver();
acessoBookserver.setLogin(autorizacao.getLogin());
acessoBookserver.setSenha(autorizacao.getSenha());

Usuario usuario = usuarioLogado();
usuario.setAcessoBookserver(acessoBookserver);

usuarios.save(usuario);
```

No trecho de código apresentado, as credenciais do usuário são mantidas no banco de dados do `client`. Isso é um problema, pois caso o usuário altere suas credenciais na aplicação `bookserver`, esse Client não conseguirá mais acessar os recursos do usuário. Ao usar OAuth 2.0 com o grant type Resource Owner Password Credentials, podemos salvar um token de acesso em vez das credenciais do usuário, e assim já não temos mais o problema da duplicidade de dados.

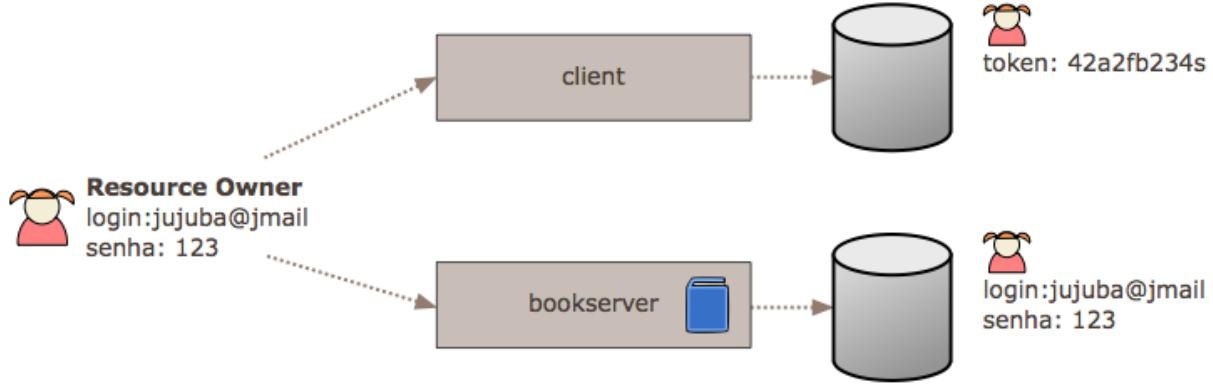


Figura 7.2: Armazenando token de acesso em vez das credenciais do usuário

Vamos partir então para um pouco de prática. Abra os projetos `client` e `bookserver` para iniciar a configuração do Authorization Server, com o grant type Resource Owner Password Credentials.

7.1 Configurando o Authorization Server

Vamos focar primeiro no projeto `bookserver`. Então, abra a classe `ConfiguracaoOAuth2` e altere o método `configure` da classe interna `OAuth2AuthorizationServer`, conforme mostrado a seguir.

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
    clients.inMemory()
        .withClient("cliente-app")
        .secret("123456")
        .authorizedGrantTypes("password")
        .scopes("read", "write")
        .resourceIds(RESOURCE_ID);
}
```

A nova versão do método `configure` agora utiliza os métodos `authorizedGrantTypes` e `scopes`. O método `authorizedGrantTypes` recebe como parâmetro a string `password`, que indica que estamos

adicionando suporte para o grant type Resource Owner Password Credentials. O método `scopes` define dois escopos que podem ser solicitados pelo Client e possivelmente autorizados pelo Resource Owner.

Escopos de um token de acesso

Os escopos são usados para indicar o que um Client pode acessar ou executar. No exemplo mostrado no código anterior, foram definidos os escopos `read` e `write` para indicar que o Client solicita acesso à leitura e à escrita nos recursos do usuário. Os escopos devem ser definidos pelo Authorization Server, e a forma como o Client descobre quais escopos estão disponíveis não é definida na especificação do protocolo OAuth 2.0.

A especificação diz que o Authorization Server deveria documentar quais escopos estão disponíveis para uso na API. Apesar de estarmos especificando os escopos no projeto de exemplo, inicialmente não validaremos se um token utilizado durante o acesso a uma API possui ou não os escopos necessários para realizar uma operação específica. Apenas tenha em mente que o Resource Server é o responsável por permitir acesso a um determinado recurso de acordo com o escopo do token sendo usado.

Nesse momento, os escopos estão sendo definidos no projeto apenas para ilustrar sua utilização no processo de solicitação do token de acesso junto ao Authorization Server. Durante a solicitação de um token de acesso, o Client pode informar os escopos que deseja, e o Authorization Server deve retornar na resposta quais deles foram autorizados para o token disponibilizado. Mais à frente, vou apresentar como usar os escopos para definir o que pode ou não ser acessado com um token de acesso.

Validação das credenciais do Resource Owner

A configuração está quase pronta, mas antes de iniciar a aplicação, é preciso informar na configuração do Authorization Server um

`AuthenticationManager`, pois a classe do Spring Security OAuth2, responsável por gerar o token de acesso no fluxo que estamos usando, precisa de uma referência para um `AuthenticationManager`. É bem justo, pois é necessário validar as credenciais do Resource Owner que está autorizando o uso de seus recursos.

O Spring Security OAuth2 disponibiliza classes específicas para gerar tokens de acesso para cada um dos grant types definidos no protocolo OAuth 2.0.

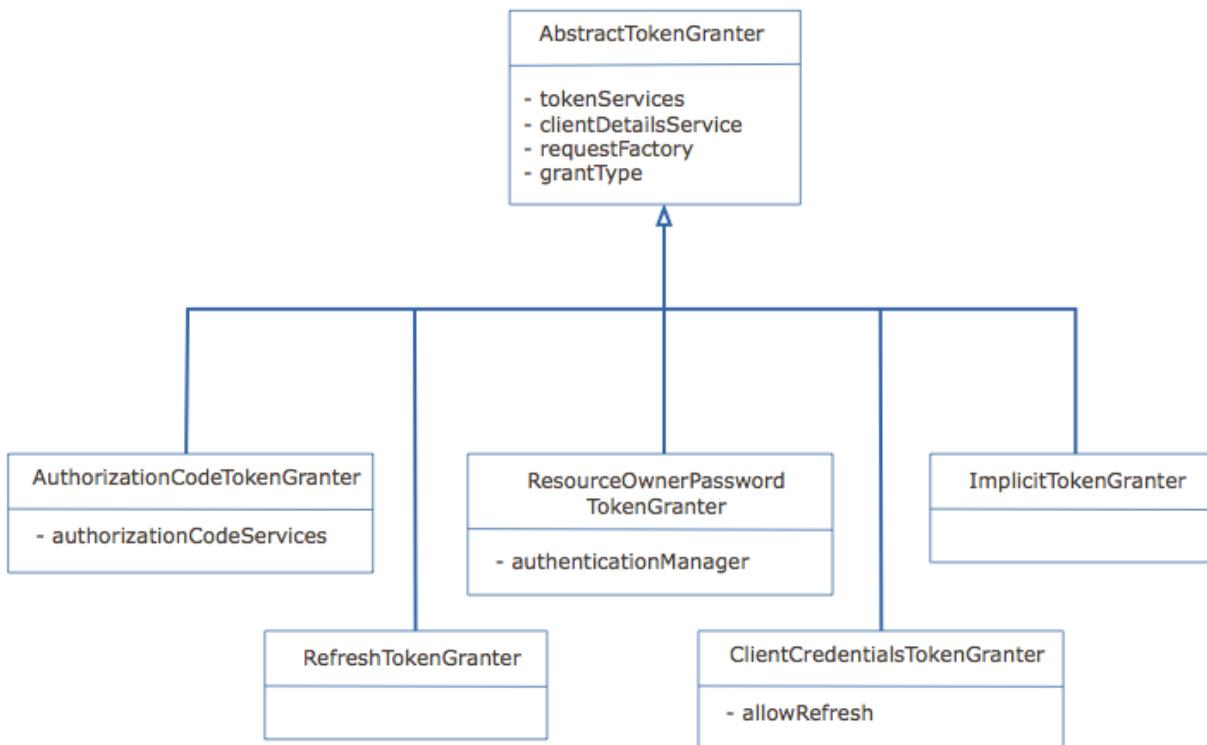


Figura 7.3: Token granters

Adicione então o seguinte trecho de código na classe interna `OAuth2AuthorizationServer` que estamos usando para configurar o Authorization Server:

```
@Autowired
private AuthenticationManager authenticationManager;

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)
```

```
    throws Exception {
        endpoints.authenticationManager(authenticationManager);
    }
}
```

Caso a configuração do `AuthenticationManager` não seja realizada, ao subir a aplicação e tentar solicitar um token de acesso, o seguinte erro será retornado:

```
{
    "error": "unsupported_grant_type",
    "error_description": "Unsupported grant type: password"
}
```

Com a configuração que adicionamos, agora a aplicação já está protegendo o endpoint `/api/v2/livros` com OAuth 2.0. Inicie a aplicação `bookserver`, mas antes de usar a API, vamos primeiro entender como montar uma requisição para obter um token de acesso e como acessar o endpoint protegido por OAuth 2.0.

7.2 Entendendo cada etapa do grant type Password

Na etapa inicial, precisamos obter as credenciais do usuário, para em seguida enviar uma requisição para o Authorization Server para obter um token de acesso. Na aplicação `client` que estamos usando como exemplo, o usuário possui uma tela específica para fornecer as credenciais da aplicação `bookserver`.

Entretanto, como a forma como o usuário fornece suas credenciais para o Client não é coberta pela especificação do protocolo OAuth 2.0, vamos manter a mesma forma de obter o login e a senha do usuário que estamos usando até agora na aplicação Client. Contudo, é preciso lembrar de que **as credenciais do usuário devem ser descartadas assim que o token de acesso for obtido**.

Já mostrei bastantes detalhes sobre as credenciais do usuário, mas e quanto ao Client? Por que este recebe um `client_id` e um `client_secret` quando é registrado no Authorization Server? Essas credenciais devem ser usadas para identificar o Client durante a solicitação do token.

De acordo com a especificação, caso o Client for do tipo confidencial ou o Authorization Server tiver gerado credenciais para o Client, então o Authorization Server deve realizar a **autenticação** do Client. No caso da aplicação `bookserver`, o Client possui um `client_id` e um `client_secret` que são `cliente-app` e `123456`, respectivamente.

A autenticação do Client será realizada por meio de `HTTP Basic Authentication`, portanto codifique a string `cliente-app:123456` com `base64`, pois precisaremos desse conteúdo para criar a URL de requisição de token. Ao codificar as credenciais da aplicação `client`, obteremos o seguinte valor: `Y2xpZW50ZS1hcHA6MTIzNDU2`.

Agora já temos dados suficientes para solicitar um token de acesso no Authorization Server. Mas qual é o endpoint que vamos usar? Conforme falado no capítulo anterior, temos o endpoint `/oauth/token` para solicitar um token de acesso.

ENDPOINTS DO AUTHORIZATION SERVER

O Authorization Server fornece endpoints para autorização e solicitação de token de acesso. No Spring Security OAuth2, são disponibilizados alguns endpoints adicionais para verificação de token, renderização de erros e um endpoint específico para uso de JWT (mais detalhes serão vistos adiante). Não se preocupe com todos os endpoints fornecidos pelo Spring Security OAuth2 agora, pois cada um será apresentado quando for necessário.

Solicitação do token de acesso

Para solicitar o token de acesso, o Client precisa enviar um POST para o endpoint `/oauth/token` informando os seguintes dados no corpo da requisição:

Parâmetro	Obrigatório	Descrição
<code>grant_type</code>	Sim	Indica o fluxo a ser usado, que nesse caso é <code>password</code> .
<code>username</code>	Sim	Login do usuário na aplicação protegida por OAuth 2.0.
<code>password</code>	Sim	Senha do usuário na aplicação protegida por OAuth 2.0.
<code>scope</code>	Não	Escopos separados por um espaço em branco.

A requisição precisa utilizar o formato `x-www-form-urlencoded` e, como o Client precisa se autenticar, precisamos usar o cabeçalho `Authorization`. No nosso caso, estamos usando o `HTTP Basic Authentication` como mecanismo de autenticação do Client, portanto as credenciais estão sendo codificadas com `Base64`. Inicie a aplicação `bookserver` e tente enviar a seguinte solicitação via `cURL`. Não esqueça de utilizar um usuário que exista na aplicação `bookserver`.

```
curl -X POST \
-H "Authorization: Basic Y2xpZW50ZS1hcHA6MTIzNDU2" \
-H "Content-Type: application/x-www-form-urlencoded" \
-H "Accept: application/json" \
-d
'grant_type=password&username=jujuba@mailinator.com&password=123&scope=rea
d write' "http://localhost:8080/oauth/token"
```

Ao executar esse comando, você deve obter uma saída parecida com a mostrada a seguir. Caso ocorra algum erro durante a execução do comando anterior, verifique se este foi digitado

corretamente. Atenção para os **espaços** existentes entre cada parâmetro do comando `CURL`.

```
{  
    "access_token": "51ec40fb-368c-4a91-8191-cfe8f5ce25b2",  
    "token_type": "bearer",  
    "expires_in": 16789,  
    "scope": "read write"  
}
```

Repare que na resposta, além do campo `access_token`, também recebemos `token_type`, `expires_in` e `scope`. Por enquanto, não se preocupe com o campo `scope`.

O atributo `token_type` indica que o token fornecido pelo Authorization Server é do tipo `Bearer`. Conforme falado anteriormente, um `Bearer` é um tipo de token específico que pode ser usado por qualquer Client que tenha posse dele. Por isso, mais uma vez, vale ressaltar a importância de manter esse tipo de token criptografado tanto quando estiver sendo trafegado pela rede (através de `TLS/SSL`) como quando estiver gravado em um banco de dados (o token também não deve ser exibido nem mesmo nos logs da aplicação).

Contudo, mesmo realizando a proteção do token de acesso através de criptografia, caso esse token ainda assim seja roubado por um Client malicioso, ele poderá obter acesso indevido aos recursos do usuário por tempo indeterminado. Para contornar esse problema, o Spring Security OAuth2 permite ao Authorization Server devolver o atributo `expires_in` que também está definido na especificação do protocolo OAuth 2.0. Esse atributo indica o tempo de validade em segundos do token de acesso. Assim que o tempo de validade do token expirar, o Client precisa obter um novo token de acesso.

Existem duas formas que o Client pode usar para recuperar o novo token de acesso. Uma forma é solicitá-lo novamente utilizando as credenciais do usuário. O problema dessa abordagem é que teremos de solicitar as credenciais do usuário novamente. Outra forma de obter um novo token é usar um *refresh token* que, por sua

vez, resolve o problema de precisar solicitar as credenciais do usuário novamente. O uso de `refresh_token` será explicado posteriormente em outro capítulo.

NOTA SOBRE SEGURANÇA

Como esse endpoint conta com autenticação do Client, é importante protegê-lo contra ataques de força bruta. Para se proteger contra esse tipo de ataque, é possível limitar o número de requisições que podem ser aceitas em um determinado intervalo de tempo (técnica conhecida como *rate limit*). Além disso, a comunicação entre o Client e o Authorization Server deve sempre ser realizada via TLS/SSL .

Utilizando o token de acesso

Agora que já obtemos um token de acesso para o Client, vamos tentar acessar o endpoint de consulta de livros que está sendo protegido por OAuth 2.0. Para isso, usaremos o seguinte comando curl .

```
curl -X GET -H "Authorization: Bearer 51ec40fb-368c-4a91-8191-cfe8f5ce25b2" \
"http://localhost:8080/api/v2/livros"
```

E a resposta deve ser uma coleção de livros do Resource Owner, parecida com a mostrada a seguir:

```
[  
  {  
    "id": 17,  
    "titulo": "Vidas secas",  
    "nota": 9  
  },  
  {  
    "id": 18,  
    "titulo": "Logicomix",  
    "nota": 10
```

```
}
```

```
]
```

Mas o que acontece se o Client tentar enviar um token de acesso inválido? Os motivos para se utilizar um token inválido são basicamente dois:

- Quando o tempo de validade do token expirou;
- Quando o token de acesso sendo utilizado simplesmente não existe mais no Authorization Server.

Em cada um dos casos, o Spring Security OAuth2 devolve um erro diferente. Quando o tempo de validade de um token expirar, o Client deve receber uma resposta parecida com a mostrada a seguir, cujo status http deve ser 401.

```
{
  "error": "invalid_token",
  "error_description": "Access token expired: d91b1973-18c8-4e4a-b04e-
0128feff1d04"
}
```

Quando se tenta utilizar um token de acesso que expirou, o Authorization Server descarta esse token de sua base de dados (mesmo utilizando o modelo em memória como estamos usando por enquanto para a aplicação bookserver). Uma vez que o token tenha sido descartado, ele passa a não ser mais reconhecido pelo Authorization Server. Logo, se o Client tentar utilizar o mesmo token, a resposta será um pouco diferente (apesar de o status http continuar sendo 401).

```
{
  "error": "invalid_token",
  "error_description": "Invalid access token: d91b1973-18c8-4e4a-b04e-
0128feff1d04"
}
```

Repare que a descrição da mensagem de erro que, antes era *Access token expired*, agora é *Invalid access token*.

7.3 Implementação do Client

Vamos agora ver como podemos implementar a solicitação e utilização do token de acesso na aplicação `client`. Para isso, abra o projeto `client` do diretório `livro-spring-oauth2/capitulo-7`. Em primeiro lugar, precisamos alterar o modelo atual, pois estamos armazenando as credenciais do usuário quando ele autoriza a exibição dos livros na linha do tempo.

Agora é necessário armazenar o token de acesso em vez das credenciais do usuário. Para isso, altere os atributos da classe `AcessoBookserver` conforme mostrado a seguir:

```
@Embeddable  
@ToString  
public class AcessoBookserver {  
  
    @Getter @Setter  
    @Column(name = "token_bookserver")  
    private String accessToken;  
  
}
```

Repare que, além de alterar o atributo na classe, também precisamos fazer algumas alterações em nosso banco de dados. Para isso, execute o seguinte comando SQL no console do MySQL. Não se esqueça de executar o comando usando o banco de dados `clientapp`.

```
ALTER TABLE usuario ADD COLUMN token_bookserver VARCHAR(60);
```

Agora vamos alterar a classe `BookserverService` que tem a responsabilidade de realizar a integração com a aplicação `bookserver`. O método `livros` vai receber um token em formato `String` em vez das credenciais do usuário. Também precisamos alterar como o cabeçalho `Authorization` é enviado, pois agora não será mais usado o `HTTP Basic Authentication`.

Como vamos usar o token OAuth 2.0, precisamos definir o cabeçalho `Authorization` como `Bearer`. Outra alteração importante a ser feita é o endpoint que vamos utilizar. Agora o endpoint usado deve ser referente à nova versão da API, protegida por OAuth 2.0, ficando então como `/api/v2/livros`.

```
public List<Livro> livros(String token) throws
UsuarioSemAutorizacaoException {
    RestTemplate restTemplate = new RestTemplate();

    MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
    headers.add("Authorization", "Bearer " + token);

    String endpoint = "http://localhost:8080/api/v2/livros";

    RequestEntity<Object> request = new RequestEntity<Object>(
        headers, HttpMethod.GET, URI.create(endpoint)
    );

    // resto do código omitido
}
```

Com o `service` pronto para buscar os livros do usuário usando OAuth 2.0, altere também o método `principal` da classe `MinhaContaController` para enviar o token de acesso registrado na conta do usuário.

```
@RequestMapping(value = "/principal")
public ModelAndView principal() {

    Usuario usuario = usuarioLogado();
    String token = usuario.getAcessoBookserver().getAccessToken();

    ModelAndView mv = new ModelAndView("minhaconta/principal");

    try {
        mv.addObject("livros", bookserverService.livros(token));
    } catch (UsuarioSemAutorizacaoException e) {
        mv.addObject("erro", e.getMessage());
    }
}
```

```
        return mv;
    }
```

Com as alterações realizadas, temos tudo pronto para que o Client possa consultar os livros do usuário utilizando um token OAuth 2.0. Porém, como o token deve ser obtido? Quando este será armazenado na nova coluna `token_bookserver` que foi criada na tabela `usuarios`?

Para facilitar o processo de obtenção do token de acesso, criei uma classe chamada `PasswordTokenService`, disponível no pacote `br.com.casadocodigo.integracao.bookserver`. Vamos dar uma olhada no método `getToken` dessa classe.

```
public class PasswordTokenService {
    public OAuth2Token getToken(String loginDoUsuario, String senhaDoUsuario) {

        RestTemplate restTemplate = new RestTemplate();
        BasicAuthentication clientAuthentication =
            new BasicAuthentication("cliente-app", "123456");

        RequestEntity<MultiValueMap<String, String>> requestEntity = new
RequestEntity<>(
            getBody(loginDoUsuario, senhaDoUsuario),
            getHeader(clientAuthentication),
            HttpMethod.POST,
            URI.create("http://localhost:8080/oauth/token")
        );

        ResponseEntity<OAuth2Token> responseEntity = restTemplate.exchange(
            requestEntity,
            OAuth2Token.class);

        if (responseEntity.getStatusCode().is2xxSuccessful()) {
            return responseEntity.getBody();
        }

        // isso deve ser tratado de forma melhor (apenas para exemplo)
```

```

        throw new RuntimeException("error trying to retrieve access token");
    }
    // getBody e getHeader omitidos
}

```

O método `getToken` realiza um `POST` para o endpoint `/oauth/token` do Authorization Server, de uma forma parecida com a que fizemos ao usarmos o comando `cURL` anteriormente. Repare que precisamos usar o login e a senha do usuário para criar o corpo da requisição através do método `getBody`, como também das credenciais do `client` para criar o `header` do `http`. Vamos configurar agora o corpo e o cabeçalho da requisição `http` para obtenção do token que são criados através dos métodos `getBody` e `getHeader`, respectivamente.

```

@Service
public class PasswordTokenService {
    private MultiValueMap<String, String> getBody(String loginDoUsuario,
String senhaDoUsuario) {
        MultiValueMap<String, String> dadosFormulario =
            new LinkedMultiValueMap<>();

        dadosFormulario.add("grant_type", "password");
        dadosFormulario.add("username", loginDoUsuario);
        dadosFormulario.add("password", senhaDoUsuario);
        dadosFormulario.add("scope", "read write");

        return dadosFormulario;
    }
}

```

Os mesmos dados de formulário que passamos no comando `cURL` também estão sendo passados agora, de forma dinâmica. Repare que os nomes dos atributos continuam sendo os mesmos, ou seja, `grant_type`, `username`, `password` e `scope`.

Além de precisar autenticar o `client`, também precisamos informar que estamos enviando um conteúdo codificado com `x-www-form-urlencoded`, e que aceitamos como resposta um conteúdo no formato

JSON . Para isso, veja como ficou a implementação do método `getHeader` :

```
@Service
public class PasswordTokenService {
    private HttpHeaders getHeader(BasicAuthentication clientAuthentication)
{
    HttpHeaders httpHeaders = new HttpHeaders();
    httpHeaders.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
    httpHeaders.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    httpHeaders.add("Authorization",
        "Basic " + clientAuthentication.getCredenciaisBase64());
    return httpHeaders;
}
}
```

A classe `PasswordTokenService` já abstrai todo o processo de solicitação do token de acesso, mas como podemos usá-la? E como guardaremos o token obtido através da utilização do método `getToken` da classe `PasswordTokenService` ?

Para obter o token, primeiro abra a classe `IntegracaoController` e injete a classe `PasswordTokenService` :

```
@Controller
@RequestMapping("/integracao")
public class IntegracaoController {
    // outros atributos omitidos
    @Autowired
    private PasswordTokenService passwordTokenService;
    // métodos omitidos
}
```

Repare que o método `autorizar` ainda está salvando as credenciais informadas pelo usuário. Remova o conteúdo do método `autorizar` e adicione o seguinte trecho de código:

```
@Controller
@RequestMapping("/integracao")
public class IntegracaoController {
    // atributos omitidos
```

```

// métodos omitidos
@RequestMapping(method = RequestMethod.POST)
public ModelAndView autorizar(Autorizacao autorizacao) {
    Usuario usuario = usuarioLogado();

    OAuth2Token token = passwordTokenService.getToken(
        autorizacao.getLogin(), autorizacao.getSenha());

    AcessoBookserver acessoBookserver = new AcessoBookserver();
    acessoBookserver.setAccessToken(token.getAccessToken());

    usuario.setAcessoBookserver(acessoBookserver);

    usuarios.save(usuario);

    return new ModelAndView("redirect:/minhaconta/principal");
}
}

```

Com a alteração realizada no método `autorizar`, agora estamos salvando o `access_token` fornecido pelo Authorization Server e as credenciais do usuário são descartadas. Dessa forma, o único dado que guardamos no Client é o token, não comprometendo mais a segurança do usuário e evitando de certa forma o acoplamento que tínhamos com as credenciais do usuário.

Agora inicie as aplicações `bookserver` e `client`, e realize alguns testes criando novos usuários e novos livros para eles. Perceba que, para o usuário final, quando utilizamos o grant type Resource Owner Password Credentials, o fluxo permaneceu igual, ou seja, a mudança que fizemos trouxe mais segurança de modo que a experiência do usuário não foi afetada.

Armazenamento do token de acesso no Client

Ao receber o token de acesso, o Client precisa mantê-lo em algum lugar seguro, pois precisará utilizar o token durante as requisições que acessam recursos do usuário. Já sabemos que o token deve

ser protegido quando transitado pela rede (através de SSL/TLS) e que deve ser protegido quando persistido.

A forma como o Client vai armazenar o token de acesso obtido não é definida pela especificação OAuth 2.0, mas o Client é livre para escolher a estratégia que for mais viável. Dentre as possíveis estratégias, o Client pode usar algoritmos de criptografia para proteger o token. No exemplo que estamos usando, não foi adicionado nenhum mecanismo de criptografia para facilitar a leitura do código. Mas em uma aplicação em produção não se esqueça de persistir esse dado de forma segura.

Armazenamento do token de acesso no Authorization Server

Uma forma interessante para proteger os tokens de acessos no banco de dados do Authorization Server é aplicar um algoritmo de `hash` no token antes de armazenar no banco de dados. Dessa forma, quando o Client enviar um token de acesso durante uma requisição, o server pode aplicar o mesmo algoritmo de `hash` no token recebido, e então comparar com o `hash` armazenado no banco de dados.

Nesse primeiro momento, estamos realizando toda configuração em memória. Porém, mais adiante quando for mostrada a configuração através de banco de dados, você poderá ver como o Spring já automatiza esse processo de criptografia.

7.4 Quando usar o grant type Password

Esse grant type deve ser usado quando existe um alto nível de confiança entre o Resource Owner e o Client, pois o Resource Owner continua tendo de disponibilizar suas credenciais para uma aplicação terceira. Esse fluxo pode ser usado em fase de migração quando já existem aplicações terceiras que utilizam as credenciais

do usuário para acessar a API (assim como no caso da aplicação `client`).

Configuramos esse grant type primeiro por ser uma opção mais simples de implementar, até porque ele não possui a fase de autorização, assim evitando o redirecionamento entre aplicações. Todo o processo para obtenção do token de acesso acontece no **back-end**, sem a interação do usuário.

Caso você esteja implementando uma aplicação nativa para mobile que acessa às APIs do seu próprio sistema (tanto mobile quanto o server fazem parte da mesma solução), esse *grant type* também pode ser interessante. Isso porque o grau de confiança do usuário em passar suas credenciais deveria ser alto o suficiente, já que tudo faz parte de um mesmo sistema. Entretanto, nesse caso, é importante que as credenciais não sejam mantidas no aplicativo do smartphone, para evitar que sejam roubadas.

Você ainda pode se perguntar: mas e as credenciais do próprio Client? Elas podem ficar armazenadas no dispositivo? Para resolver esse tipo de problema, existe uma especificação que se trata de uma extensão do OAuth 2.0, que define meios para registro automático do Client. Através do seu registro automático, as credenciais são geradas dinamicamente de modo que o Client pode manter o `client_id` e o `client_secret` na memória em vez de manter no código-fonte do aplicativo.

Apesar de existirem cenários nos quais se justifique o uso do grant type apresentado nesse capítulo, vale lembrar de que ele deve ser evitado sempre que possível, pois o principal objetivo ao se utilizar OAuth 2.0 é que o Resource Owner não precise compartilhar suas credenciais com o Client.

7.5 Conclusão

Esse foi o primeiro capítulo em que pudemos interagir com um fluxo OAuth 2.0 para obter um token de acesso e para consumir um endpoint protegido por OAuth 2.0. De agora em diante, os capítulos são mais práticos. Você passa a entender os detalhes de cada grant type e escreve um pouco de código.

Além de entender os detalhes do protocolo, é importante que entenda também quando um determinado grant type deve ser usado. Utilizar o grant type inadequado pode comprometer a segurança da sua aplicação e os dados do usuário.

Apesar da prática envolvida neste capítulo, alguns desenvolvedores podem ficar curiosos sobre como o Resource Server protege o endpoint. Como tudo isso acontece por dentro do Spring Security OAuth2? Vamos para o próximo capítulo, que mostrará justamente isso.

CAPÍTULO 8

Funcionamento interno do Spring Security OAuth2

Este pequeno capítulo é para o desenvolvedor que está curioso em saber como o Spring Security OAuth2 funciona internamente durante a tentativa de acesso a um recurso protegido por OAuth 2.0. Caso não estiver com tanta curiosidade, você pode pular para o próximo capítulo. Porém, é interessante conhecer um pouco dos detalhes do framework, pois dessa forma será mais fácil compreender como você pode customizar alguns pontos do Spring Security OAuth2.

Através de uma visão de alto nível, vamos entender como a requisição é interceptada pelo Spring Security OAuth2.

Lembra de quando falei no tópico sobre *Configuração do Resource Server*, que durante a configuração do **Resource Server** o Spring Security OAuth2 adiciona o filtro `OAuth2AuthenticationProcessingFilter` na cadeia de filtros do Spring Security (cadeia de filtros que é representada pela classe `FilterChainProxy`)? Pois é, esse filtro já está sendo utilizado quando tentamos acessar os recursos protegidos por OAuth 2.0.

Uma requisição que é enviada para acessar um recurso protegido por OAuth 2.0 é interceptada por uma cadeia de filtros, conforme mostrado a seguir:

`WebAsyncManagerIntegrationFilter`

`SecurityContextPersistenceFilter`

`HeaderWriterFilter`

`LogoutFilter`

`OAuth2AuthenticationProcessingFilter`

`RequestCacheAwareFilter`

`SecurityContextHolderAwareRequestFilter`

`AnonymousAuthenticationFilter`

`SessionManagementFilter`

`ExceptionTranslationFilter`

`FilterSecurityInterceptor`

Figura 8.1: Cadeia de filtros usando OAuth 2.0

Quando o filtro `OAuth2AuthenticationProcessingFilter` é invocado, o token enviado na requisição é extraído do header `Authorization` e inicia-se o fluxo de validações em cima desse token. Todo o processo de validação acontece dentro de um bloco `try-catch` que, em caso de falha, lança uma exceção do tipo `OAuth2Exception`.

O bloco de código mais importante durante o processo de validação do token dentro do `OAuth2AuthenticationProcessingFilter` é mostrado a seguir:

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {

    final HttpServletRequest request = (HttpServletRequest) req;
    final HttpServletResponse response = (HttpServletResponse) res;

    try {

        Authentication authentication = tokenExtractor.extract(request);

        if (authentication == null) {
            // código omitido, pois não nos
            // interessa quando um token não é enviado
        }
        else {
            // mais código omitido
            Authentication authResult = authenticationManager
                .authenticate(authentication);

            eventPublisher.publishAuthenticationSuccess(authResult);

            SecurityContextHolder.getContext().setAuthentication(authResult);
        }
    }
    catch (OAuth2Exception failed) {
        SecurityContextHolder.clearContext();
        // código omitido
        return;
    }
}
```

```

        }

        chain.doFilter(request, response);
    }
}

```

Uma boa parte do código do filtro do Spring Security OAuth2 foi omitido, pois nesse momento o interesse é mostrar como o framework executa o processo de validação do token de acesso, e de autenticação do **Client**. Perceba que, no final, caso tudo tenha dado certo, o próximo filtro da cadeia é invocado para dar continuidade ao fluxo de validações.

O processo de autenticação do Client acontece quando o filtro delega a autenticação para o objeto `authenticationManager`, que é do tipo `OAuth2AuthenticationManager`. Mas como o `authenticationManager` consegue identificar se o token informado existe e é válido? Veja o que acontece internamente através do fluxo apresentado na próxima figura.

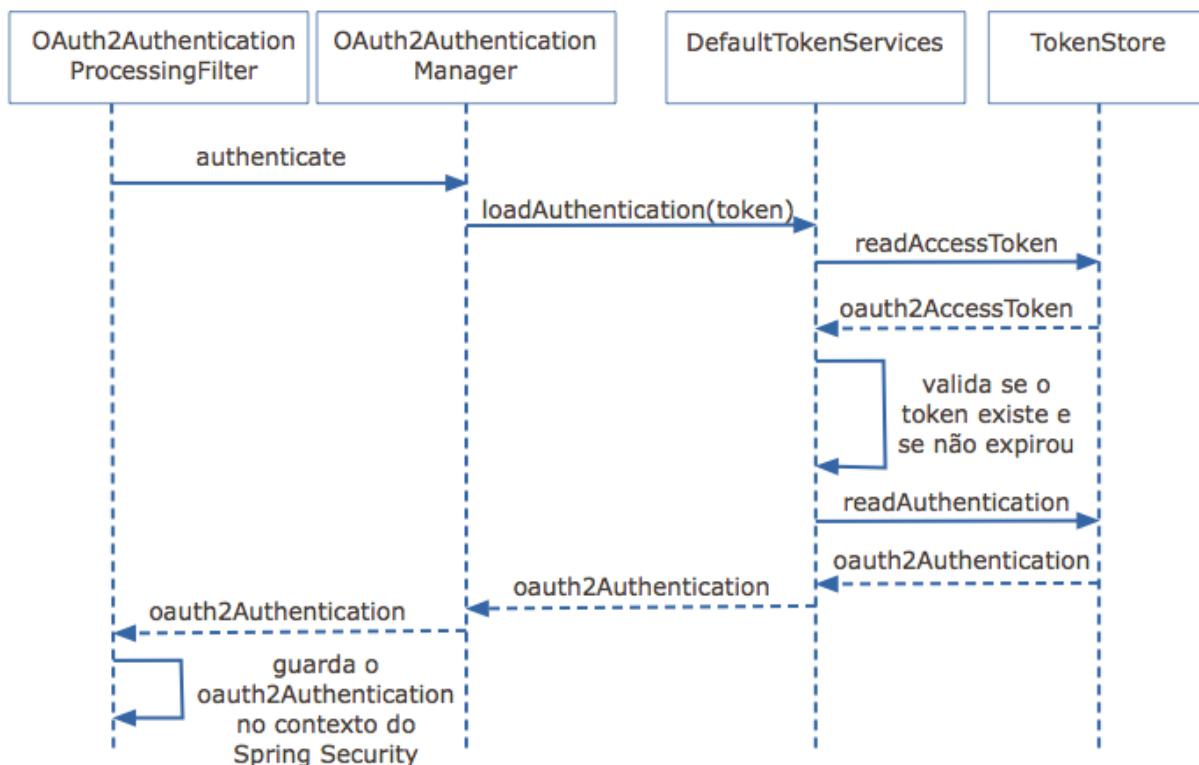


Figura 8.2: Fluxo de autenticação do token de acesso

Como podemos ver na figura anterior, temos três entidades essenciais para o processo de autenticação do token de acesso enviado pelo Client. No fluxo apresentado, temos as classes `OAuth2AuthenticationManager`, `DefaultTokenServices` e a interface `TokenStore`.

A classe `DefaultTokenServices` implementa as interfaces `AuthorizationServerTokenServices`, `ResourceServerTokenServices`, `ConsumerTokenServices` e `InitializingBean`. Em meio a tantas interfaces implementadas, o importante é que você perceba que essa é a classe responsável pela criação de token de acesso, criação de refresh tokens, recuperação de tokens, além de outras funcionalidades que não vou citar agora.

Muito do trabalho que a classe `DefaultTokenServices` realiza conta com a colaboração de um `TokenStore`. O `TokenStore` representa o repositório onde são mantidos os tokens de acesso criados para os Clients registrados no nosso Authorization Server. No projeto `bookserver`, configuramos a aplicação para manter os dados do Client na memória. Por conta disso, a implementação que está sendo utilizada nesse momento é `InMemoryTokenStore`.

Veja na figura a seguir quais tipos de repositórios de token o Spring Security OAuth2 já disponibiliza por padrão:

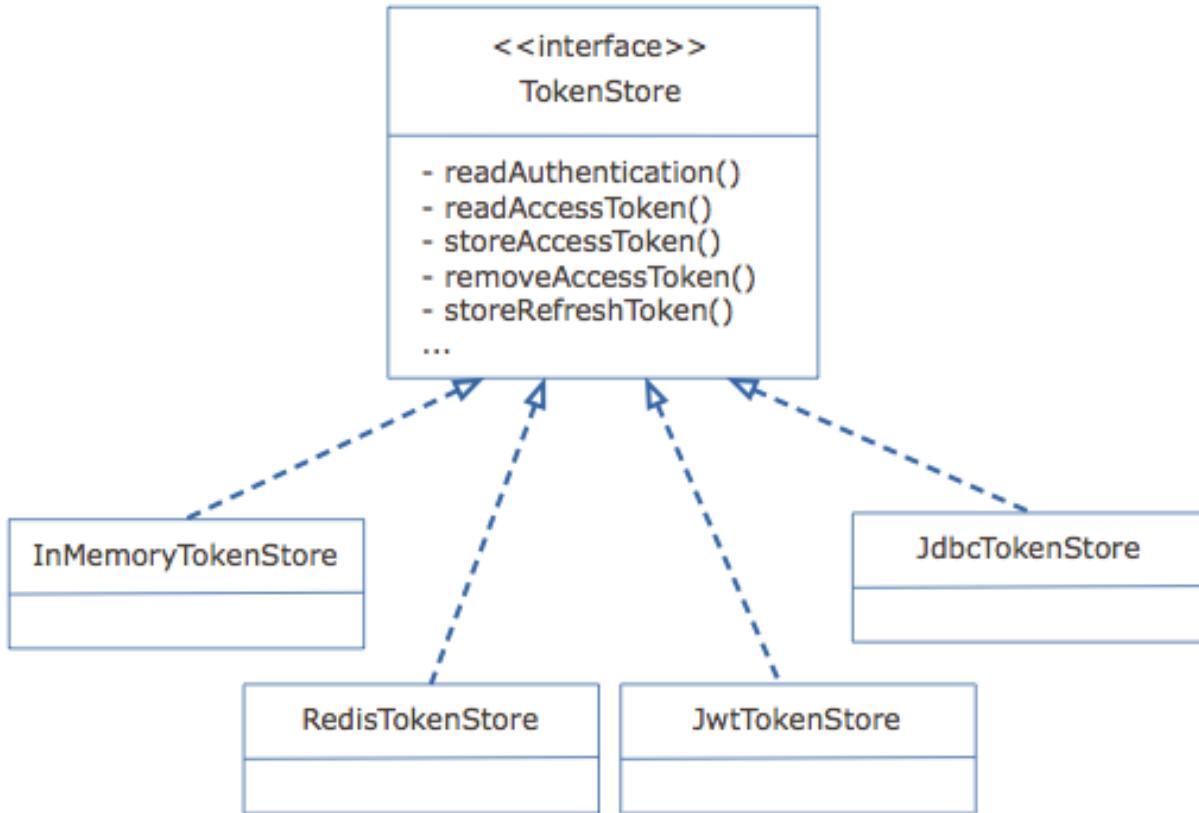


Figura 8.3: Tipos de token store do Spring Security OAuth2

Assim como muitos componentes do Spring Security, um `TokenStore` pode ser customizado, permitindo ao desenvolvedor implementar uma outra forma de armazenar os tokens de acesso. Como exemplo, você poderia implementar um `TokenStore` baseado em outros bancos de dados `NoSQL`, além do `Redis`. Você poderia até mesmo criar uma customização para armazenar os tokens de acesso em outra aplicação.

Voltando ao fluxo de autenticação que acontece dentro do filtro do Spring Security OAuth2, caso o token de acesso enviado na requisição seja válido, ele é guardado dentro do contexto de segurança do Spring Security através da seguinte linha de código:

```
SecurityContextHolder.getContext().setAuthentication(authResult);
```

8.1 Acesso aos recursos de usuário específico

Na aplicação `bookserver`, queremos que quando o Client usar um token de acesso para buscar os livros do usuário, a resposta contenha apenas os livros do usuário associado ao token sendo utilizado. Agora que você conhece como o token de acesso é armazenado na aplicação e no contexto do Spring Security, você também sabe como recuperar os dados do usuário associado ao token de acesso enviado durante a requisição. Na aplicação `bookserver`, basta usar o código seguinte para isso:

```
Authentication authentication = SecurityContextHolder.getContext()
    .getAuthentication();
ResourceOwner donoDosLivros = (ResourceOwner)
authentication.getPrincipal();
```

E por sinal, é exatamente isso que já estamos fazendo no método que recupera o Resource Owner dentro da classe `LivrosApiController`.

```
private Usuario donoDosLivros() {
    Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
    ResourceOwner donoDosLivros = (ResourceOwner)
authentication.getPrincipal();
    return usuarios.buscarPorID(donoDosLivros.getId());
}
```

Perceba como o Spring Security dá a flexibilidade necessária para permitir que o Spring Security OAuth2 seja adicionado sem precisar alterar a forma como interagimos com a camada de segurança. Estamos recuperando os dados do usuário autenticado da mesma forma que fazíamos quando estávamos usando *HTTP Basic Authorization*.

8.2 Conclusão

Após ler este capítulo, você fica sabendo um pouco sobre o que acontece por trás do framework Spring Security OAuth2 e entende mais como ele se integra com o Spring Security. O legal de conhecer um pouco do que está acontecendo por dentro de um framework ou biblioteca é que fica mais fácil agir quando surgir uma necessidade de customização no projeto em que você estiver trabalhando.

Neste capítulo, apresentamos como o Spring Security OAuth2 faz para autenticar um token de acesso que chega para um endpoint protegido por OAuth 2.0. Além disso, também vimos como os dados do Client e do Resource Owner foram mantidos no contexto do Spring Security, facilitando a identificação de ambos dentro da aplicação.

CAPÍTULO 9

Authorization Code grant type

Agora vamos explorar o fluxo OAuth 2.0 mais usado. Para obtenção de tokens de acesso, esse fluxo conta com o processo de **autorização** do usuário e se trata da solução clássica para o problema da delegação de acesso a recursos protegidos. Através dele, o Resource Owner pode **autorizar** um Client a acessar recursos em seu nome sem precisar compartilhar suas credenciais em nenhum momento.

O *grant type* já abordado, **Resource Owner Password Credentials**, ainda repassa as credenciais do usuário, resolvendo parcialmente os problemas encontrados quando um usuário deseja delegar acesso aos seus recursos para uma aplicação terceira. Vamos entender como funciona o fluxo **Authorization Code** para autorização e solicitação do token de acesso.

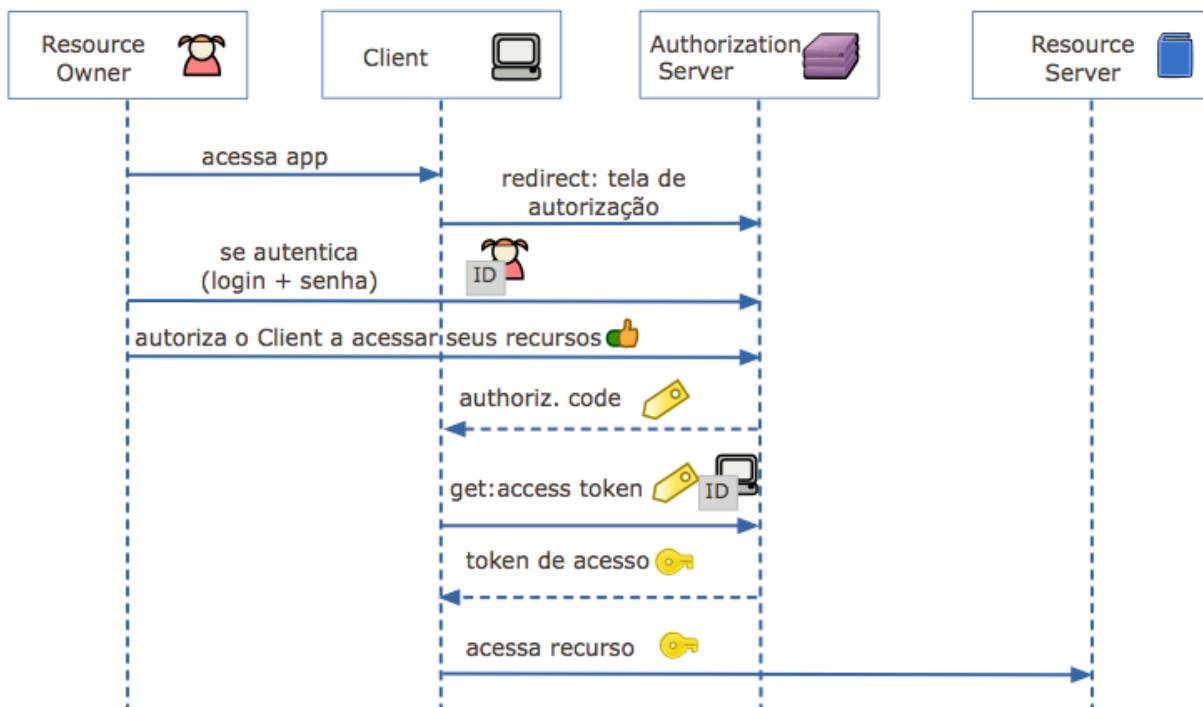


Figura 9.1: Authorization Code grant type

9.1 Visão geral das etapas do grant type Authorization Code

Conforme podemos ver na figura anterior, o fluxo possui uma etapa a mais em relação ao grant type Resource Owner Password Credentials. Agora além de solicitar o token de acesso, existe a etapa na qual o usuário autoriza o Client a acessar seus recursos. Para que o usuário possa autorizar o acesso, o Client **redireciona** o usuário para o endpoint de autorização disponível no Authorization Server.

Ao acessar esse endpoint, o Authorization Server solicita a **autenticação do Resource Owner** e, em seguida, apresenta as permissões solicitadas pelo Client para o usuário. As permissões mostradas para o Resource Owner são diretamente relacionadas aos escopos informados pelo Client no endpoint de autorização.

Caso o Resource Owner autorize o acesso aos seus recursos, o Authorization Server redireciona-o de volta para o Client. Mas como o Authorization Server sabe o endereço para onde redirecionar o Resource Owner?

O Authorization Server utiliza para isso a URL de redirecionamento que pode ter sido informada durante o registro do Client, ou através de parâmetro no próprio endpoint de autorização. Caso o Authorization Server não obrigue o Client a registrar uma URL de redirecionamento, então torna-se obrigatório que o Client envie esse parâmetro no endpoint de autorização. O ideal é que ambas estratégias sejam usadas para que, dessa forma, o Authorization Server possa validar a URI de redirecionamento.

Durante o redirecionamento para o Client, o Authorization Server precisa adicionar mais um parâmetro na URL. Esse parâmetro é o `authorization_code`, que contém uma chave que deve ser usada pelo Client na etapa de solicitação do token de acesso. O `authorization_code` deve ser utilizado apenas uma vez e deve possuir

um tempo de validade bem curto para evitar que ele seja roubado e usado por um usuário mal-intencionado para obter um token de acesso.

O tempo de validade recomendado pela especificação do protocolo OAuth 2.0 é de, no máximo, 10 minutos. Outro ponto definido na especificação é que, caso o `authorization_code` seja usado mais de uma vez, recomenda-se que o Authorization Server **cancele** todos tokens gerados com base nesse `authorization_code`. Além do parâmetro `authorization_code`, o Authorization Server também pode devolver o `state` caso o Client tenha passado esse parâmetro no endpoint de autorização. Usar o parâmetro `state` também é muito importante para evitar um ataque CSRF.

De posse do `authorization_code`, o Client já pode solicitar o token de acesso, sendo necessário para isso se autenticar no Authorization Server. Caso o Client tenha sido autenticado, o Authorization Code esteja correto e a URI de redirecionamento seja igual à registrada pelo Client, então o Authorization Server devolve um token de acesso como resposta, podendo também devolver um `refresh_token`.

Assim que o token for obtido, o Client pode então acessar os recursos do usuário da mesma forma que fizemos quando utilizamos o *grant type* Resource Owner Password Credentials.

9.2 Configurando o Authorization Server

Para acompanhar o código apresentado neste capítulo, importe os projetos `bookserver` e `client` contidos no diretório `livro-spring-oauth2/capitulo-9`. Em primeiro lugar, vamos adicionar o suporte ao *grant type* **Authorization Code** do lado do Authorization Server. Para isso, abra a classe `ConfiguracaoOAuth2`, para editarmos a classe interna `OAuth2AuthorizationServer`.

Na configuração de grant types, que é realizada através da chamada ao método `authorizedGrantTypes`, além da string `password` que já está sendo passada como parâmetro, adicione também o valor `authorization_code` conforme mostrado a seguir.

```
@EnableAuthorizationServer
protected static class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.inMemory()
            .withClient("cliente-app")
            .secret("123456")
            .authorizedGrantTypes("password", "authorization_code")
            .scopes("read", "write")
            .resourceIds(RESOURCE_ID);
    }

}
```

Com o novo argumento que estamos passando para o método `authorizedGrantTypes`, o Authorization Server já passa a dar suporte ao **grant type Authorization Code**. A partir de agora, já podemos interagir com o Authorization Server. Porém, antes de iniciarmos os testes, é preciso saber como realizar as solicitações de autorização e de token de acesso.

9.3 Etapa de autorização

Nessa etapa, precisamos simplesmente redirecionar o usuário para o endpoint de autorização que, no caso da aplicação `bookserver`, é `http://localhost:8080/oauth/authorize`. Para isso, veremos quais parâmetros precisamos passar na URI de autorização.

Parâmetro	Obrigatório	Descrição
<i>response_type</i>	Sim	Esse parâmetro deve conter o valor <i>code</i> .

Parâmetro	Obrigatório	Descrição
<i>client_id</i>	Sim	Indica o Client que está solicitando autorização do usuário.
<i>redirect_uri</i>	Não	Indica uma URI do Client que deve ser usada para receber o <i>authorization_code</i> caso o Resource Owner autorize o acesso aos seus recursos. Caso o Client não tenha registrado uma URI de redirecionamento durante o cadastro, ele é obrigado a passar a URI no endpoint de autorização utilizando a codificação <i>application/x-www-form-urlencoded</i> .
<i>scope</i>	Não	Representa os escopos sendo solicitados pelo Client. Apesar de não ser obrigatório, a especificação deixa em aberto a possibilidade de obrigar o envio do parâmetro <i>scope</i> durante a fase de autorização. No caso do Spring Security OAuth2, caso o Client não informe os escopos, estes são recuperados da configuração do Client e utilizados por padrão.

Parâmetro	Obrigatório	Descrição
<code>state</code>	Não	Esse parâmetro, apesar de não ser obrigatório, é recomendado para evitar ataques CSRF. Seu conteúdo deve ser uma <i>string</i> que o Authorization Server deve devolver na resposta, para que o Client possa validar se o <i>authorization_code</i> obtido realmente foi solicitado. Essa é uma validação simples que apenas compara a string enviada com a recebida.

Para começarmos um teste simples, inicie a aplicação `bookserver` e acesse a seguinte URL através do *browser*:

```
http://localhost:8080/oauth/authorize?client_id=cliente-
app&response_type=code&redirect_uri=http://localhost:9000/callback
```

CODIFICAÇÃO DA URI DE REDIRECIONAMENTO

No link de autorização apresentado, o parâmetro `redirect_uri` não está codificado com o formato *URL encoded*. Nesse exemplo, não codifiquei a URL apenas para facilitar a leitura, mas lembre-se de que a especificação solicita a codificação dela.

Você pode realizar a codificação ou decodificação através do site <http://www.url-encode-decode.com/>. Ao aplicar a codificação da URL `http://localhost:9000/callback`, obteremos o seguinte resultado: `http%3A%2F%2Flocalhost%3A9000%2Fcallback`.

Ao acessar a URL informada, você deve receber como resposta do Authorization Server a seguinte tela de login, pois, conforme já foi

dito, o **Resource Owner** precisa se autenticar no Authorization Server.



Figura 9.2: Autenticação do Resource Owner

Nesse teste bem simples você já pode perceber uma grande vantagem de se utilizar OAuth 2.0 como solução para delegação de acesso. Em vez de fornecer o login e a senha para o Client, o Resource Owner vai se autenticar diretamente no Authorization Server.

Informe o login e a senha do usuário que você cadastrou na aplicação bookserver e você deverá ver a tela a seguir:

OAuth Approval

Do you authorize 'cliente-app' to access your protected resources?

- scope.read: Approve Deny
- scope.write: Approve Deny

[Authorize](#)

Figura 9.3: Autorização do Resource Owner

A tela de autorização gerada pelo próprio Spring Security OAuth2 (que pode ser customizada) apresenta para o Resource Owner os escopos solicitados pelo Client e o nome do Client que está solicitando o acesso aos recursos. Selezionando as opções Approve , o Authorization Server vai redirecionar o Resource Owner para o endpoint `http://localhost:9000/callback` (que foi enviado na requisição para o endpoint de autorização) com o parâmetro `code` adicionado na URL. **Guarde** o código que você receber, pois vamos usá-lo mais adiante para obter um token de acesso.

Veja a seguir um exemplo de resposta:

`http://localhost:9000/callback?code=zmgX24`

Ao executar esse teste, conseguimos ver o conteúdo do parâmetro `code` , porque esse endpoint não existe na aplicação `client` e o browser não navega para tela alguma. Em um fluxo comum, o Resource Owner não tem condições de ver esse conteúdo sendo trafegado através da URL, a menos que use para isso ferramentas especiais como `Wireshark` , que permite analisar pacotes trafegados na rede.

Também é possível utilizar um plugin do Firefox chamado `NoRedirect` , que faz com que um redirecionamento não seja

realizado automaticamente para um determinado endereço. Dessa forma, o usuário tem a chance de decidir se autoriza ou não o redirecionamento e, nesse meio tempo, pode analisar o conteúdo dos parâmetros adicionados na URI de redirecionamento.

Vamos fazer apenas mais um teste com o endpoint de autorização. Dessa vez, vamos adicionar o parâmetro `state` para saber se o Spring Security OAuth2 já retorna o valor desse parâmetro na resposta da autorização. Para fazer o teste, acesse a seguinte URL (deslogue o usuário caso ele ainda esteja logado na aplicação `bookserver`).

```
http://localhost:8080/oauth/authorize?client_id=cliente-
app&response_type=code&redirect_uri=http://localhost:9000/callback&state=teste
```

Ao realizar esse teste em minha máquina, o Authorization Server me redirecionou para o endpoint de callback do Client, contendo os seguintes parâmetros:

```
http://localhost:9000/callback?code=kXXPR9&state=teste
```

A resposta trouxe o conteúdo do parâmetro `state` exatamente como esperávamos. Apesar de esse ser um comportamento exigido pela especificação, muitas empresas ainda hoje não tomam esse cuidado. Não suportar o parâmetro `state` faz com que o Client se torne vulnerável, comprometendo os recursos do Resource Owner. No capítulo *Algumas considerações sobre segurança*, vou mostrar como é fácil realizar um ataque CSRF em um servidor que não valida o parâmetro `state`.

Erros durante a etapa de autorização

Mostrei um exemplo no qual o usuário autoriza o acesso aos recursos. Mas e quando o usuário **não autoriza**? Como o Client fica sabendo disso? Nessa situação, o Authorization Server retorna o parâmetro `error=access_denied` na URI de redirecionamento, podendo informar também o motivo através do parâmetro `error_description`.

No caso de erro por **acesso negado**, o Authorization Server deve informar o Client assim como acontece na aplicação `bookserver`. Porém caso algum erro ocorra por causa de problemas com a URI de redirecionamento ou com `client_id`, o Authorization Server **não** deve redirecionar o usuário de volta para o Client.

Os problemas com o parâmetro `redirect_uri` podem ser por conta de estar digitado incorretamente, por ser inválido ou simplesmente por não ter sido especificado no endpoint de autorização. Já os problemas com o `client_id` podem ocorrer caso seja informado um `client_id` que não exista na base do Authorization Server.

A tabela a seguir mostra a lista dos possíveis erros que podem ocorrer na fase de autorização.

Erro	Descrição
<code>invalid_request</code>	Falta algum parâmetro que é obrigatório, ou algum parâmetro foi informado de forma incorreta. Nesse caso, o usuário não é redirecionado de volta para o Client.
<code>invalid_client</code>	Retornado quando usamos o Spring Security OAuth2 e um Client não registrado tenta solicitar autorização do usuário. Nesse caso, o usuário também não é redirecionado de volta para o Client.

Erro	Descrição
<code>unauthorized_client</code>	O Client não pode solicitar o Authorization Code com o método HTTP usado.
<code>access_denied</code>	O Resource Owner não deu autorização para o Client acessar seus recursos.

Erro	Descrição
<i>unsupported_response_type</i>	O Authorization Server não suporta o grant type informado.
<i>invalid_scope</i>	Indica que o Client tentou solicitar um escopo que não é permitido para ele.
<i>server_error</i>	Indica que um erro interno ocorreu no Authorization Server. Esse parâmetro é importante, pois o Authorization Server não tem condições de devolver um erro 500 quando um erro interno ocorre. Isso é porque, como é usado um redirecionamento, o status http retornado é diferente (nesse caso, algum 3xx).
<i>temporarily_unavailable</i>	Indica que o Authorization Server não pode atender à solicitação no momento. Esse parâmetro é usado pelo mesmo motivo do erro interno. Um erro 503 também não pode ser retornado para o Client via redirecionamento.

O Authorization Server também pode devolver mais dois parâmetros que são `error_description` e `error_uri`. O parâmetro `error_description` fornece uma descrição básica do erro e o parâmetro `error_uri` deve fornecer uma URI que possa ser utilizada pelo desenvolvedor do Client para obter mais detalhes sobre o erro ocorrido.

9.4 Etapa de solicitação do token

Uma vez que o Resource Owner já tenha autorizado o acesso aos seus recursos, o Client já pode solicitar um token de acesso para o Authorization Server. Para solicitar o token de acesso, o Client precisa enviar uma requisição HTTP contendo os parâmetros vistos a seguir no corpo da requisição. Como a solicitação do token é realizada usando o método `POST`, os parâmetros devem ser enviados no formato `application/x-www-form-urlencoded`.

Parâmetro	Obrigatório	Descrição
<code>grant_type</code>	Sim	Esse parâmetro deve conter o valor <code>authorization_code</code> .
<code>code</code>	Sim	Deve conter o código recebido através do parâmetro <code>code</code> como resposta da etapa de autorização na URL de callback.
<code>redirect_uri</code>	Depende	É obrigatório caso o parâmetro <code>redirect_uri</code> tenha sido enviado na solicitação de autorização. Caso esse conteúdo tenha sido enviado na etapa de autorização, a <code>redirect_uri</code> deve ser a mesma na etapa de solicitação do token.
<code>client_id</code>	Depende	Indica o Client que está solicitando o token de acesso. É obrigatório caso o Client não esteja sendo autenticado pelo Authorization Server.

Além dos parâmetros enviados no corpo da requisição, o Authorization Server deve autenticar o Client caso ele seja do tipo **confidencial**, ou o Client possua credenciais (`client_id` e

`client_secret`). Realizar a autenticação do Client traz os seguintes benefícios:

- Caso o Client tenha as credenciais roubadas, é mais fácil alterar as credenciais do Client ou desativá-lo do que ter de revogar o acesso a todos os refresh tokens gerados para o Client em questão.
- Fica mais fácil implementar um esquema de rotação de credenciais do Client para reforçar a segurança do OAuth Provider. Imagine ter de fazer rotação dos `refresh_tokens` gerados para cada Client. Caso o número de Clients seja grande, isso pode ser um problema.

Agora vamos realizar um teste para saber se a aplicação `bookserver` está realmente permitindo a geração do token de acesso. Como estamos usando autenticação do Client através de `HTTP Basic Authentication`, então precisamos codificar as credenciais com Base 64.

Codifique as credenciais `cliente-app:123456` com Base64 e adicione o atributo `Authorization` no cabeçalho `HTTP` da requisição. Além disso, adicione os dados necessários no corpo da requisição conforme mostrado a seguir.

```
curl -X POST -H "Authorization: Basic Y2xpZW50ZS1hcHA6MTIzNDU2" \
-H "Content-Type: application/x-www-form-urlencoded" \
-d
'grant_type=authorization_code&redirect_uri=http://localhost:9000/callback
&scope=read write&code=BoEHea' "http://localhost:8080/oauth/token"
```

Atenção para o campo `code` usado no corpo da requisição. No meu caso, o valor desse campo foi `BoEHea`, mas no seu caso o valor deverá ser **diferente**. É preciso enviar o valor obtido na URL de redirecionamento quando o Resource Owner autorizou o acesso ao Client.

Caso tudo esteja certo, o Authorization Server deve devolver uma resposta parecida com a seguinte:

```
{  
    "access_token": "9b0a07ae-78cc-4aaa-9883-e7f8e9b4ab52",  
    "token_type": "bearer",  
    "expires_in": 42279,  
    "scope": "read write"  
}
```

Se você tentar utilizar um `authorization_code` que já tenha sido usado para obter um token de acesso, o Authorization Server devolve a seguinte resposta de erro:

```
{  
    "error": "invalid_grant",  
    "error_description": "Invalid authorization code: kXXPR9"  
}
```

9.5 Utilizando o token de acesso

Para acessar os recursos do usuário, podemos realizar a mesma requisição que fizemos quando usamos o grant type Resource Owner Password Credentials. A forma como um token de acesso foi obtida não importa.

```
curl -X GET -H "Authorization: Bearer 9b0a07ae-78cc-4aaa-9883-  
e7f8e9b4ab52" \  
"http://localhost:8080/api/v2/livros"
```

Ao enviar a requisição, você deve receber uma coleção de livros do Resource Owner parecida com a mostrada a seguir:

```
[  
    {  
        "id": 17,  
        "titulo": "Vidas secas",  
        "nota": 9  
    },  
    {  
        "id": 18,
```

```
        "titulo": "Logicomix",
        "nota": 10
    }
]
```

9.6 Implementação do Client

Implementar o Client utilizando o grant type **Authorization Code** já é mais interessante por conta dos redirecionamentos que acontecem nesse fluxo. Para que a aplicação Client possa acessar os recursos do usuário usando o grant type Authorization Code, vamos editar o projeto `client` contido no diretório `livro-spring-oauth2/capitulo-9`.

Até agora, quando o usuário acessa a tela `Minha Conta`, ele tem a opção de mostrar os livros que estão cadastrados na aplicação `bookserver`, bastando clicar no link <http://localhost:9000/integracao>. Ao clicar nesse link, o usuário é levado a uma tela na qual ele fornece suas credenciais para que a aplicação Client possa repassar para o Authorization Server durante a fase de solicitação de token.

Esse processo é realizado através de um *forward* implementado no método `integracao` da classe `IntegracaoController`:

```
@Controller
@RequestMapping("/integracao")
public class IntegracaoController {
    // atributos omitidos

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView integracao() {
        return new ModelAndView("minhaconta/integracao");
    }
    // outros métodos omitidos
}
```

O fluxo que acabei de descrever é adequado para o grant type Resource Owner Password Credentials. Entretanto, como queremos usar o grant type Authorization Code, é necessário alterar o método `integracao` para que seja feito um **redirecionamento** para o Authorization Server.

Para facilitar a integração com a aplicação `bookserver` através do grant type **Authorization Code**, já disponibilizei no projeto `client` uma classe responsável pela geração do endpoint de autorização e solicitação do token. Antes de implementar a nova versão do método `integracao`, vamos injetar uma instância de `AuthorizationCodeTokenService` na classe `IntegracaoController`.

```
@Controller
@RequestMapping("/integracao")
public class IntegracaoController {
    // outros atributos omitidos
    @Autowired
    private AuthorizationCodeTokenService authorizationCodeTokenService;
    // métodos omitidos
}
```

E agora que estamos injetando essa classe, vamos alterar o método `integracao` da classe `IntegracaoController`, conforme mostrado no trecho de código a seguir:

```
@RequestMapping(method = RequestMethod.GET)
public ModelAndView integracao() {
    String endpointDeAutorizacao =
    authorizationCodeTokenService.getAuthorizationEndpoint();
    return new ModelAndView("redirect:" + endpointDeAutorizacao);
}
```

Abra a classe `AuthorizationCodeTokenService` e veja como funciona o método `getAuthorizationEndpoint`. É interessante ver como os parâmetros são adicionados na URL e como a URI de redirecionamento é codificada para poder ser enviada.

```
BasicAuthentication clientAuthentication =
    new BasicAuthentication("cliente-app", "123456");
```

```

// recuperando os dados necessários
// para o endpoint de autorização

String endpointDeAutorizacao = "http://localhost:8080/oauth/authorize";

// montando a URI
Map<String, String> parametros = new HashMap<>();
parametros.put("client_id",
getEncodedUrl(clientAuthentication.getLogin()));
parametros.put("response_type", "code");
parametros.put("scope", getEncodedUrl("read write"));
parametros.put("redirect_uri",
getEncodedUrl("http://localhost:9000/integracao/callback"));

return construirUrl(endpointDeAutorizacao, parametros);

```

Repare que estamos codificando todos os parâmetros para que eles sejam formatados adequadamente na URL de autorização. Outro detalhe importante nessa configuração é o parâmetro `redirect_uri` que estamos definindo. Esse endpoint de callback ainda precisa ser criado no `client` para que este possa obter o `authorization_code` e, em seguida, solicitar o token de acesso.

Ao subir a aplicação e tentar acessar o link para mostrar os livros da aplicação `bookserver`, você deve ser redirecionado automaticamente para a tela de login do Authorization Server. Caso queira realizar esse teste agora, não se esqueça de deixar nulo o campo `token_bookserver` da tabela `usuario` no banco de dados do `client`. Caso contrário, o Client vai usar o token existente para recuperar os livros e você não conseguirá testar o fluxo **Authorization Code**. Se você prosseguir com a autorização, o Client vai falhar quando receber o `authorization_code` na URI de callback, pois ainda não temos um `controller` que responde a esse endpoint.

Altere o nome do método `autorizar` da classe `IntegracaoController` para `callback`, para que faça sentido com o nome do endpoint de callback que vamos definir. Repare que também adicionei o atributo `value` da anotação `@RequestMapping` com o valor `/callback`. Dessa

forma, esse método passa a tratar as requisições que chegam em `/integracao/callback`, que é justamente o endpoint que configuramos no método `getAuthorizationEndpoint` da classe `AuthorizationCodeTokenService`.

Veja como ficou a definição do método de callback. **Não esqueça de alterar o método http suportado de POST para GET:**

```
@RequestMapping(value = "/callback", method = RequestMethod.GET)
public ModelAndView callback(String code, String state) {
    // códigos omitidos
}
```

Agora, dentro do método, vamos alterar a forma como o token é obtido. Em vez de usar `passwordTokenService` para obter o token, vamos usar o novo objeto que estamos injetando:

```
authorizationCodeTokenService .
```

```
OAuth2Token token = authorizationCodeTokenService.getToken(code);
```

Como você pode notar, agora não usamos mais nada de credenciais do usuário. Em vez disso, estamos usando um código que representa a **autorização** dada pelo Resource Owner para o Client poder acessar recursos em seu nome.

O código da classe `IntegracaoController` já está pronto, entretanto ainda falta um detalhe no código que obtém o token de acesso. Precisamos configurar os parâmetros que são usados na solicitação do token de acesso. Abra a classe `AuthorizationCodeTokenService` e veja como está o método `getToken`.

```
public OAuth2Token getToken(String authorizationCode) {
    RestTemplate restTemplate = new RestTemplate();
    BasicAuthentication clientAuthentication = new
BasicAuthentication("cliente-app", "123456");

    RequestEntity<MultiValueMap<String, String>> requestEntity = new
RequestEntity<>(
        getBody(authorizationCode),
        getHeader(clientAuthentication),
```

```

        HttpMethod.POST,
        URI.create("http://localhost:8080/oauth/token")
    );

    ResponseEntity<OAuth2Token> responseEntity = restTemplate.exchange(
        requestEntity,
        OAuth2Token.class);

    if (responseEntity.getStatusCode().is2xxSuccessful()) {
        return responseEntity.getBody();
    }

    // isso deve ser tratado de forma melhor (apenas para exemplo)
    throw new RuntimeException("error trying to retrieve access token");
}

```

Esse método está muito parecido com o `getToken` da classe `PasswordTokenService`. Porém, agora em vez de receber as credenciais, ele recebe o argumento `authorizationCode`. Além disso, esse é o argumento que repassamos para o método privado `getBody`. Repare que todo o resto continua igual.

Vamos adicionar as seguintes configurações no método `getBody`:

```

dadosFormulario.add("grant_type", "authorization_code");
dadosFormulario.add("code", authorizationCode);
dadosFormulario.add("scope", "read write");
dadosFormulario.add("redirect_uri",
    "http://localhost:9000/integracao/callback");

```

Repare também que o método `getHeader` continua igual, pois ainda precisamos **autenticar** o Client no Authorization Server.

```

private HttpHeaders getHeader(BasicAuthentication clientAuthentication) {
    HttpHeaders httpHeaders = new HttpHeaders();

    httpHeaders.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
    httpHeaders.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    httpHeaders.add("Authorization", "Basic " +
clientAuthentication.getCredenciaisBase64());
}

```

```
    return httpHeaders;
}
```

Com a configuração finalizada, você já deve conseguir seguir o fluxo completo de autorização e solicitação do token de acesso. Como exercício, navegue na aplicação e acompanhe as etapas mostradas na figura apresentada no início deste capítulo.

É bem interessante adicionar alguns *breakpoints* na aplicação e acompanhar como os dados são trocados entre o Client e o Authorization Server. Isso ajuda bastante a entender o fluxo como um todo.

9.7 Quando usar o grant type Authorization Code

O grant type *Authorization Code* deve ser usado geralmente quando se trata de uma aplicação web que tem a capacidade de **redirecionar** o usuário para o Authorization Server, para que este possa autorizar o uso de seus recursos. Além disso, ele é indicado para aplicações **confidenciais**, ou seja, que têm a capacidade de proteger o token de acesso recebido do Authorization Server.

O grant type **Authorization Code** também pode ser utilizado em aplicações nativas que têm a capacidade de disponibilizar um *browser* para que o usuário possa se autenticar no Authorization Server e autorizar o Client. Em aplicativos nativos que rodam em smartphones, é possível registrar um URL schema, de modo que sempre que uma requisição HTTP chegar para tal URL, ela é interceptada de alguma forma invocando o código que consegue tratar a requisição. Isso facilita o recebimento do `authorization_code` durante o callback de resposta da autorização.

Qualquer tipo de aplicação que não permita que o usuário seja redirecionado entre Authorization Server e Client **não** deve utilizar o grant type Authorization Code. Como exemplo de aplicações onde o

grant type Authorization Code não faz sentido, podemos citar aplicações em que o Client e o OAuth Provider fazem parte de uma mesma solução, ou onde o Client é dono dos recursos que deseja acessar.

Para esses exemplos, poderiam ser utilizados os grant types Resource Owner Password Credentials e Client Credentials, respectivamente. O grant type Resource Owner Password Credentials já conhecemos, mas o grant type Client Credentials será abordado em um capítulo posterior.

9.8 Conclusão

Este capítulo apresentou o fluxo OAuth 2.0 que deu origem ao protocolo. Entendendo o funcionamento desse *grant type*, fica mais fácil entender como funcionam os próximos *grant types* que são `implicit` e `client-credentials`. Ao estudar este capítulo, você avançou no conhecimento sobre como o Spring Security OAuth2 funciona e já pode ver como foi simples adaptar a aplicação `bookserver` para dar suporte ao grant type **Authorization Code**.

Você também viu aqui detalhes importantes quando mostrei como os dados são enviados em cada etapa e os tipos de respostas que podemos obter. Entender que nem todos os tipos de erros precisam ser mostrados nas telas do próprio Authorization Server, sem redirecionar o usuário de volta para o Client, é importante para evitar problemas de segurança. Além disso, também é preciso dar importância para os parâmetros `state` e `redirect_uri` que, atualmente, devido à má compreensão deles, diversas brechas de segurança podem ser introduzidas na aplicação.

CAPÍTULO 10

Implicit grant type para aplicações no Browser

O *grant type* que vamos estudar neste capítulo possui uma característica bem peculiar. O fluxo para obtenção do token de acesso conta apenas com a fase de autorização. Mas então como o token é obtido? Assim como no *grant type Authorization Code*, o Resource Owner também é redirecionado para a página de autorização.

Entretanto, quando ele se autentica no Authorization Server e autoriza o uso de seus recursos, o Authorization Server redireciona o usuário de volta para o Client, adicionando o token de acesso na URI de redirecionamento. Portanto, o token é entregue ao Client sem que este faça a solicitação diretamente, ou seja, o token é entregue de maneira **implícita**. Por isso o nome do *grant type* é **Implicit**.

Apesar de termos uma etapa a menos no *grant type Implicit*, a forma como o token é extraído da URL de redirecionamento pode parecer um pouco confusa inicialmente. A figura a seguir pode ajudar a entender melhor como o fluxo funciona.

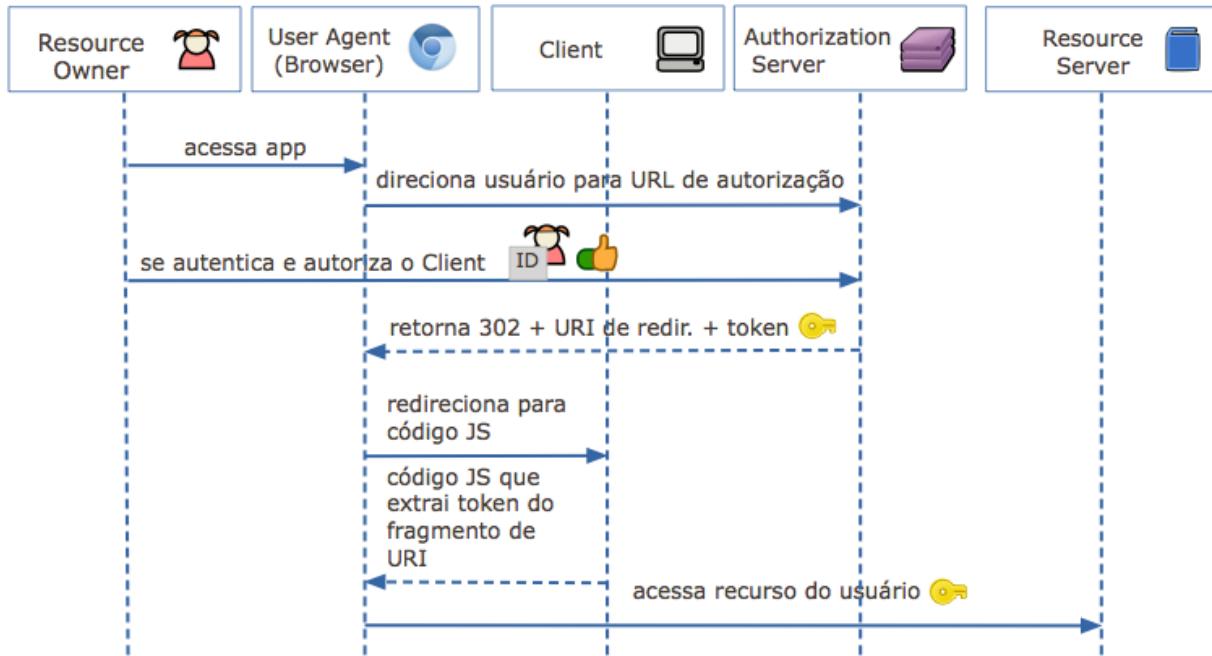


Figura 10.1: Implicit grant type

A imagem mostra uma aplicação que executa diretamente no *Browser* redirecionando o usuário para a URL de autorização do Authorization Server. No Authorization Server, o usuário se autentica, autoriza o acesso aos seus recursos e, em seguida, o Authorization Server devolve uma resposta de redirecionamento, ou seja, uma resposta com status 302 contendo a URI de redirecionamento concatenada com um fragmento de URL. É nesse fragmento que é adicionado o parâmetro `access_token`.

FRAGMENT IDENTIFIER, OU FRAGMENTO DE URL

Um *fragment identifier*, conforme definido pela RFC 3986 (<https://tools.ietf.org/html/rfc3986#section-3.5>), pode ser utilizado como componente de uma URI, permitindo identificar partes secundárias de um recurso primário. Fragmentos são adicionados na URL utilizando um caractere `#`, de modo que tudo o que estiver à direita desse caractere é um fragmento.

Um fragment identifier também pode ser alterado dinamicamente através de um código JavaScript, permitindo modificar a URL sem causar atualização na página. Apesar de não atualizar a página, o *user agent* mantém histórico quando um fragment identifier é alterado. Esse recurso é bastante usado em aplicações web ricas que não fazem atualização completa das páginas.

Uma característica interessante é que, ao realizar uma requisição que contém um fragmento, este não é enviado para o servidor. Isso é importante para o Implicit grant type no momento em que é feito o redirecionamento para a URL de callback com o fragmento adicionado pelo servidor de autorização, pois apesar de o fragmento estar disponível na URL que aparece na barra de navegação, ele não é enviado para o servidor através da URI de redirecionamento.

Perceba que, durante esse fluxo, o Client não precisa se autenticar no Authorization Server. Por conta disso, o grant type Implicit conta com o registro de uma URI de redirecionamento por parte do Client e também com a presença do usuário.

De acordo com a especificação, o Authorization Server é obrigado a solicitar o registro da URI de redirecionamento para Clients públicos ou confidenciais que usem o grant type Implicit. Entretanto, a implementação do Spring Security OAuth2 não obriga o registro da URI de redirecionamento.

No capítulo *Algumas considerações sobre segurança*, apresento os problemas que podemos encontrar caso a URI de redirecionamento não seja registrada para o Client nesse tipo de situação e como se prevenir de algumas brechas de segurança.

Como esse fluxo conta com a presença do Resource Owner, não faz sentido que o Authorization Server gere um `refresh_token` assim como nos outros fluxos. Portanto, conforme explicado na especificação do protocolo OAuth 2.0, o Authorization Server **não** deve gerar `refresh_token` para o Client quando usamos o grant type **Implicit**.

Agora como é que o token é extraído quando o Authorization Server redireciona o Resource Owner para o Client? Veja que, na figura que mostra o fluxo do grant type **Implicit**, o redirecionamento faz com que o endpoint de callback do Client seja requisitado e que, por sua vez, pode retornar um código JavaScript capaz de extrair o token que está disponível no fragmento da URL. O fragmento de URL pode ser recuperado da seguinte forma através de código JavaScript:

```
var fragment = window.location.hash;
```

Ao extrair o fragmento, o código em JavaScript pode realizar alguma lógica para identificar o token de acesso contido na variável `fragment`. Com o token de acesso disponível, basta começar a realizar as requisições para o Resource Server.

Lembre-se de que esse tipo de token de acesso deve ter um tempo de validade curto. Como o grant type **Implicit** não suporta refresh tokens, quando o token de acesso não for mais válido, o usuário deve ser redirecionado novamente para o Authorization Server, seguindo o mesmo fluxo de autorização. Para melhorar a experiência do usuário, evitando que ele tenha de passar por todo o fluxo novamente, o Authorization Server pode utilizar cookies para identificar que ele já autorizou determinado Client anteriormente.

10.1 Configurando o Authorization Server

Vamos agora configurar o Authorization Server para dar suporte ao grant type Implicit. Para realizar as configurações e os testes, importe os projetos bookserver e client do diretório livro-spring-oauth2/capitulo-10 . Abra o projeto bookserver e vamos analisar a classe ConfiguracaoOAuth2.0Auth2AuthorizationServer .

Até o momento, o Authorization Server já está dando suporte aos grant types password e authorization_code . Para adicionar suporte ao grant type Implicit, basta passar mais um argumento para o método authorizedGrantTypes , contendo a string implicit .

Note que não estamos registrando uma URI de redirecionamento do Client que, segundo a especificação, deveria ser obrigatória para o grant type Implicit. Em um sistema real, não podemos nos esquecer de obrigar o Client a registrar uma redirect_uri .

```
@EnableAuthorizationServer
protected static class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer
endpoints) throws Exception {
        endpoints.authenticationManager(authenticationManager);
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.inMemory()
            .withClient("cliente-app")
            .secret("123456")
            .authorizedGrantTypes(
                "password",
```

```

        "authorization_code",
        "implicit")
.scopes("read", "write")
.resourceIds(RESOURCE_ID);
}
}

```

Continuamos usando as mesmas configurações de *client details*, escopos e resource ID. Entretanto, precisamos realizar uma modificação na configuração do Resource Server. A configuração realizada na classe `OAuth2ResourceServer` para proteger a API de consulta de livros precisa suportar requisições AJAX cuja origem seja outro domínio.

Conforme veremos mais adiante, vamos alterar a aplicação `client` para solicitar um token e acessar a API de consulta de livros utilizando AJAX. E como a aplicação `client` executa a partir de um outro domínio, estamos tratando justamente de um exemplo de requisição `cross-domain`. Para suportar esse tipo de acesso nas APIs da aplicação `bookserver`, adicione o método `cors` na configuração do Resource Server, conforme mostrado a seguir.

```

@EnableResourceServer
protected static class OAuth2ResourceServer extends
ResourceServerConfigurerAdapter {
    // outros métodos omitidos
    @Override
    public void configure(HttpSecurity http) throws Exception {

        http
            .authorizeRequests()
            .anyRequest().authenticated().and()
            .requestMatchers()
            .antMatchers("/api/v2/**").and()
            .cors();
    }
}

```

CORS (*CROSS-ORIGIN RESOURCE SHARING*)

O mecanismo conhecido como CORS permite adicionar controle de acesso a requisições *cross-domain*, ou seja, requisições cuja origem seja de um domínio diferente da aplicação que está fornecendo um recurso. Esse recurso pode ser um endpoint de uma API, que é acessado através de AJAX (*Asynchronous Javascript and XML*).

Os *browsers* modernos já usam CORS para diminuir os riscos existentes em requisições *cross-domain*. Quando usamos o objeto `XMLHttpRequest` ou `Fetch` para enviar requisições AJAX, o *browser* automaticamente realiza primeiro uma requisição `HTTP` usando o método `OPTIONS`, para descobrir se uma requisição *cross-domain* é permitida pelo servidor. Essa requisição contém uma série de cabeçalhos que precisam ser entendidos pelo *server*.

Ao adicionar o método `cors` na configuração de segurança do Resource Server, estamos então tornando a aplicação `bookserver` capaz de processar requisições *cross-domain*. Para mais detalhes sobre o mecanismo CORS, confira o link https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

Apesar de ser adicionado o suporte ao mecanismo `CORS` na aplicação, ainda precisamos configurar o endpoint para aceitar requisições *cross-domain* utilizando a anotação `@CrossOrigin`. Através dessa anotação, podemos dizer quais **origens** podem realizar uma requisição para o endpoint.

No nosso caso, adicione o suporte *cross-domain* para qualquer origem, conforme mostrado a seguir no método `livros` da classe `LivrosApiController`.

```

@RestController
@RequestMapping({"/api/livros", "/api/v2/livros"})
public class LivrosApiController {

    @Autowired
    private Usuarios usuarios;

    @CrossOrigin
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<?> livros() {
        // código omitido
    }

    // resto do código omitido
}

```

Já podemos agora testar o funcionamento do *grant type* configurado. Porém, vamos primeiro entender quais os parâmetros necessários para realizar a requisição de autorização utilizando o *grant type* Implicit.

10.2 Etapa de autorização e solicitação do token

Temos agora uma única etapa para obter a autorização do usuário e o token de acesso. Para isso, precisamos montar a URL de solicitação do token de forma implícita, contendo os seguintes parâmetros. Lembre-se de que os parâmetros precisam ser enviados usando o formato `application/x-www-form-urlencoded`.

Parâmetro	Obrigatório	Descrição
<code>response_type</code>	Sim	Esse parâmetro deve conter o valor <i>token</i> .

Parâmetro	Obrigatório	Descrição
<i>client_id</i>	Sim	Indica o Client que está solicitando a autorização do usuário.
<i>redirect_uri</i>	Não	Não é obrigatório porque o Client já é obrigado a registrar uma URI de redirecionamento no Authorization Server. Entretanto, utilizar esse parâmetro vai reforçar a segurança, uma vez que o Authorization Server deverá validar se o <i>redirect_uri</i> informado é o mesmo que foi registrado.
<i>scope</i>	Não	Representa os escopos sendo solicitados pelo Client.
<i>state</i>	Não	Esse parâmetro, apesar de não ser obrigatório, é recomendado para evitar ataques CSRF. Seu conteúdo deve ser uma <i>string</i> que o Authorization Server deve devolver na resposta, para que o Client possa validar se o <i>token</i> obtido realmente foi solicitado. Essa é uma validação simples que apenas compara a string enviada com a recebida.

Para iniciar um teste simples, inicie a aplicação `bookserver` e acesse a seguinte URL através do *browser*.

```
http://localhost:8080/oauth/authorize?  
response_type=token&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fimplicit&  
client_id=cliente-app&scope=read+write&state=teste
```

Perceba que estamos informando o parâmetro `redirect_uri`, pois como não registramos uma URI de redirecionamento para o Client, o Authorization Server precisa saber para qual URL devolver o access token. Se não informarmos nada no nosso exemplo, o Authorization Server apresentará um erro assim que o Resource Owner fizer o login.

Ao tentar acessar a URL informada, o Resource Owner deve se autenticar da mesma maneira realizada quando solicitamos a autorização do usuário através do grant type Authorization Code.



Figura 10.2: Autenticação do Resource Owner

O Resource Owner também precisa autorizar os acessos da mesma forma que fizemos usando o grant type Authorization Code.

OAuth Approval

Do you authorize 'cliente-app' to access your protected resources?

- scope.read: Approve Deny
- scope.write: Approve Deny

Authorize

Figura 10.3: Autorização do Resource Owner

E caso o Resource Owner autorize o acesso aos seus recursos, o Authorization Server vai devolver a seguinte URL de redirecionamento (provavelmente com um `access_token` diferente).

`http://localhost:9000/implicit#access_token=812ca2c4-23c2-4712-92da-5a58ad9971b3&token_type=bearer&state=teste&expires_in=43199`

Perceba que todos os atributos que recebemos em formato JSON, quando fizemos a solicitação do token através do endpoint `/oauth/token`, estão presentes como parâmetros na própria URL de redirecionamento (como um fragmento de URL). O que precisamos fazer agora é extrair o `access_token` e começar a consultar os livros do usuário na aplicação `bookserver`.

Para acessar os recursos do usuário, basta usar o token da mesma forma que já estamos usando nos exemplos anteriores. Conforme falei no capítulo anterior, não importa como o token de acesso foi obtido.

10.3 Erros durante a etapa de autorização

Caso o usuário não autorize o acesso aos seus recursos, o Authorization Server devolve os atributos que representam o erro na forma de fragmento de URL, conforme mostrado a seguir:

```
http://localhost:9000/callback#error=access_denied&error_description=User%20denied%20access&state=teste
```

Os tipos de erros que podem ser retornados são os mesmos que já foram explicados no capítulo *Authorization Code grant type*, portanto não entrarei em detalhes sobre eles aqui.

10.4 Implementação do Client

Como exemplo de implementação do Client, vamos alterar a aplicação `client` modificando um código HTML que já adicionei e que interage com a aplicação `bookserver`, utilizando JavaScript para isso. Imagine que precisamos criar um protótipo de tela onde os dados não são processados no servidor como temos feito até agora.

O código HTML que usaremos, além de mostrar os livros que o usuário tem na aplicação `bookserver`, também deve ter o JavaScript que extrai o token de acesso do fragmento de URL. Para que esse HTML seja exibido, precisaremos criar também um endpoint de callback específico.

Abra o projeto `client` do diretório `livro-spring-oauth2/capitulo-10` e vamos editar a classe `IntegracaoController`. Vamos adicionar outro serviço, agora com a responsabilidade de gerar o endpoint de autorização com o grant type **Implicit**. Vamos então injetar na classe `IntegracaoController` um atributo que referencia a classe `ImplicitTokenService`.

```

@Controller
@RequestMapping("/integracao")
public class IntegracaoController {
    // outros atributos omitidos

    @Autowired
    private ImplicitTokenService implicitTokenService;

    // métodos omitidos
}

```

A classe `ImplicitTokenService` monta a URL de autorização, conforme mostrado a seguir:

```

@Service
public class ImplicitTokenService {

    public String getAuthorizationEndpoint() {
        BasicAuthentication clientAuthentication =
            new BasicAuthentication("cliente-app", "123456");

        String endpointDeAutorizacao =
"http://localhost:8080/oauth/authorize";

        // montando a URI
        Map<String, String> parametros = new HashMap<>();
        parametros.put("response_type", "token");
        parametros.put("client_id",
            getEncodedUrl(clientAuthentication.getLogin()));
        parametros.put("redirect_uri",
            getEncodedUrl("http://localhost:9000/integracao/implicit"));
        parametros.put("scope", getEncodedUrl("read write"));

        return construirUrl(endpointDeAutorizacao, parametros);
    }

    // métodos privados omitidos
}

```

A forma como estamos montando a URL de autorização é bastante parecida com a forma que fizemos para montar a URL de

autorização para o grant type **Authorization Code**. A diferença aqui é que estamos usando o valor `token` para o parâmetro `response_type` e a URI de redirecionamento foi trocada de `/integracao/callback` para `/integracao/implicit`.

Vamos então alterar o método `integracao` da classe `IntegracaoController` para usar o endpoint de autorização do grant type **Implicit**.

```
@RequestMapping(method = RequestMethod.GET)
public ModelAndView integracao() {
    String endpointDeAutorizacao =
implicitTokenService.getAuthorizationEndpoint();
    return new ModelAndView("redirect:" + endpointDeAutorizacao);
}
```

A aplicação já está pronta para começar a usar o grant type **Implicit**, porém vamos criar mais um método de callback que responderá à URI `/integracao/implicit` conforme mostrado a seguir:

```
@RequestMapping(value = "implicit", method = RequestMethod.GET)
public ModelAndView implicit() {
    return new ModelAndView("minhaconta/bookserver");
}
```

Como podemos ver, esse método simplesmente vai retornar uma view que se trata justamente do código HTML que contém o JavaScript que consegue extrair o token de acesso do fragmento da URI de redirecionamento. Veja a seguir um trecho do arquivo `bookserver.html` criado dentro de `resources/templates/minhaconta`, no projeto `client` (nesse trecho de código, omiti bastante conteúdo para focarmos no essencial).

```
<!DOCTYPE html>
<html>
<head>
    <title>bookserver-client</title>
</head>
<body>
    <div class="row livros">
```

```

        </div>
    </body>

<script src="/jquery.min.js"></script>
<script src="/oauth.js"></script>
<script src="/livros.js"></script>

<script>
$(function() {
    var tokenResponse = oauth.getTokenResponse();
    $('.access-token').text(tokenResponse.access_token);
    $('.token-type').text(tokenResponse.token_type);
    $('.expires-in').text(tokenResponse.expires_in);

    // código que busca os livros via ajax
    livros.getTitulos(tokenResponse.access_token, function($div) {
        $('.livros').append($div);
    });
});
</script>
</html>

```

Basicamente, quando o redirecionamento acontecer após a autorização do usuário, a página `bookserver.html` será carregada, e o código JavaScript adicionado no final do arquivo vai extrair o token de acesso e vai usá-lo para obter os livros do usuário através de uma requisição AJAX. Veja o código-fonte do arquivo `oauth.js` que extrai o token de acesso do fragmento da URI de redirecionamento.

```

$(function() {
    if (window.oauth == undefined) { window.oauth = {} };

    oauth.getTokenResponse = function() {
        var fragment = window.location.hash;
        var atributos = fragment.slice(1).split('&');
        var resposta = {};

        $(atributos).each(function(idx, atributo) {
            var chaveValor = atributo.split('=');
            resposta[chaveValor[0]] = chaveValor[1];
        });
    };
});

```

```
});

    return resposta;
}

});
```

No exemplo de código que estamos usando, o token de acesso é usado para consultar os livros do usuário diretamente através do *browser* do usuário. O token de acesso também poderia ser enviado para o servidor da própria aplicação `client`, para que ela guardasse o token do usuário para usar posteriormente.

Para ver como ficou a integração entre os sistemas, inicie as aplicações `client` e `bookserver`, e acesse a aplicação `client` através do *browser*. Autentique-se com algum usuário que você já tenha cadastrado e veja como ficou a integração com a aplicação `bookserver`.

Ao clicar no link para a tela que mostra os livros do usuário, você deve se autenticar no Authorization Server, autorizar os escopos solicitados e então deverá ver a seguinte tela. Sua URL deve ser parecida com

http://localhost:9000/integracao/implicit#access_token=8cd40e0a-bb3d-40b7-9f44-d72feb724a32&token_type=bearer&expires_in=43199.

Client

Olá jujuba@jmail

Dados extraídos do fragmento de url

access_token	8cd40e0a-bb3d-40b7-9f44-d72feb724a32
token_type	bearer
expires-in	41195

Livros

Design Patterns

Voltar

Figura 10.4: Tela utilizando Implicit grant type

Perceba que essa tela está mostrando os detalhes sobre o token obtido quando o usuário autorizou o acesso aos seus recursos. Obviamente esse não é um exemplo de aplicação segura. Esses dados estão sendo exibidos na tela apenas para mostrar que, através do código JavaScript, foi possível extrair todos atributos recebidos no fragmento de URI.

10.5 Quando usar o grant type Implicit

O grant type **Implicit** deve ser usado em situações nas quais o Client executa inteiramente no browser. Aplicações criadas com HTML e JavaScript que interagem com o servidor de forma dinâmica através de requisições AJAX, por exemplo, fazem todo o sentido usar o grant type **Implicit**. Caso a aplicação web seja processada por um servidor web ou uma aplicação nativa, o grant type mais indicado é o **Authorization Code** por questões de segurança.

10.6 Conclusão

Neste capítulo, além do grant type **Implicit**, você também aprendeu mais um mecanismo de segurança conhecido como CORS. Com os grant types apresentados até agora, você já pode implementar soluções de integração em qualquer cenário que envolva a delegação de acesso de um Resource Owner para um Client. No próximo capítulo, você vai conhecer outro grant type que, em vez de permitir o acesso aos recursos de um Resource Owner, permite ao Client acessar recursos em benefício próprio.

CAPÍTULO 11

Client Credentials

Todos os grant types estudados até agora possuíam algo em comum, que é o acesso do Client aos recursos do Resource Owner. Mas imagine uma aplicação que protege suas APIs com OAuth 2.0 e deseja fornecer serviços para aplicações internas que não precisam realizar ações em benefício de um usuário específico.

Pensando na aplicação `bookserver`, um exemplo de cenário seria um Client interno usado pelos administradores do sistema, que possui como requisito consultar o total de livros cadastrados na aplicação `bookserver`. Como são sistemas diferentes, acessar a API da aplicação `bookserver` é uma solução bem melhor do que compartilhar o banco de dados.

O PROBLEMA DE COMPARTILHAR BANCO DE DADOS

Quando compartilhamos um banco de dados em diversas aplicações, o primeiro problema que temos é quando uma modificação no banco afeta diversas aplicações. Isso dificulta a manutenção do sistema como um todo. Além disso, ao colocar um sistema em produção, considerando mudanças em banco de dados compartilhado, é preciso sincronizar o deploy de cada aplicação, tornando o processo mais lento do que deveria. Evite o compartilhamento de bancos de dados entre sistemas sempre que puder.

No cenário que pensamos para a aplicação `bookserver`, seria interessante que o acesso por parte desse Client administrativo também continuasse sendo realizado através de token OAuth 2.0. Essa é justamente uma situação em que o Client precisa acessar recursos em benefício próprio, sendo a solução ideal utilizar o grant type **Client Credentials**.

Através do grant type Client Credentials, o Client pode solicitar um token de acesso sem nenhum vínculo com algum usuário existente no **OAuth Provider**. Veja na figura a seguir o fluxo completo para solicitação de token.

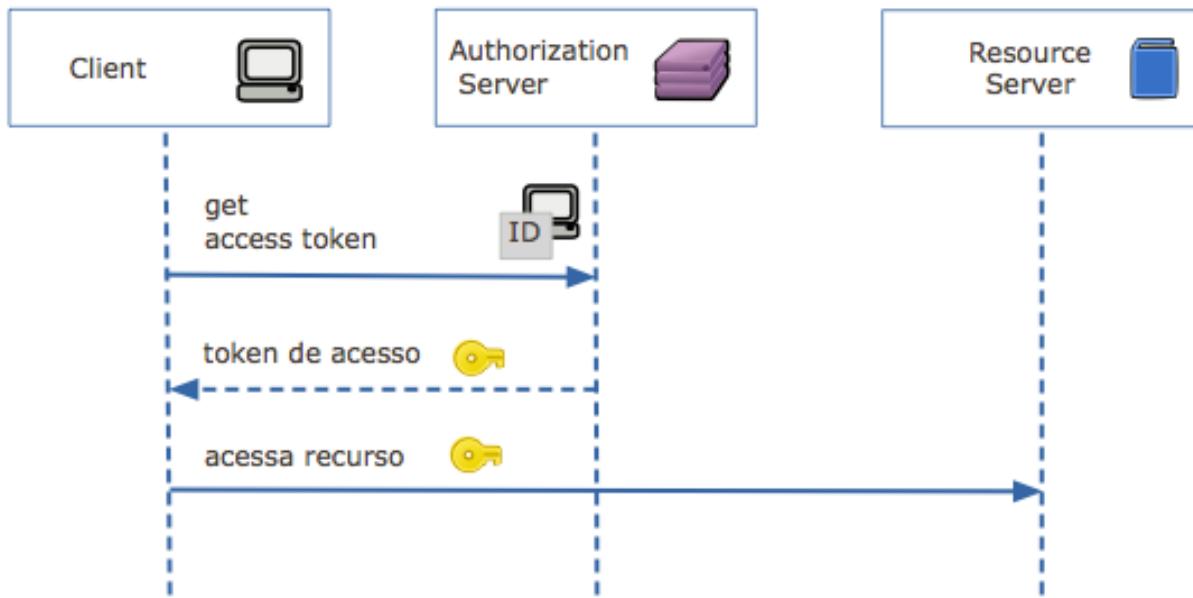


Figura 11.1: O grant type Client Credentials

Podemos observar que o fluxo para obter o token de acesso usando o grant type **Client Credentials** é muito simples. Existe apenas a etapa de solicitação do token de acesso usando as credenciais do próprio Client, ou seja, o `client_id` e o `client_secret`. Uma vez obtido o token, a utilização é realizada da mesma forma que é feita quando se obtém o token através dos outros grant types.

11.1 Implementação do novo serviço para aplicações internas

Para ilustrar o uso do grant type **Client Credentials**, vamos primeiro criar um novo *endpoint* na aplicação `bookserver` que retornará o total de livros cadastrados na base de dados. Os exemplos criados para

este capítulo estão disponíveis no diretório `livro-spring-oauth2/capitulo-11`. Importe os projetos `bookserver` e `admin` para a sua IDE.

No projeto `bookserver`, crie a classe `AdministracaoController` no pacote `br.com.casadocodigo.livros.api`, conforme mostrado a seguir:

```
@Controller
@RequestMapping("/api/v2/admin")
public class AdministracaoController {

    @Autowired
    private RepositorioDeLivros repositorioDeLivros;

    @RequestMapping(method = RequestMethod.GET, value = "/total_livros")
    public ResponseEntity<Long> getTotalDeLivros() {
        Long totalDeLivros = repositorioDeLivros.getTotalDeLivros();
        return new ResponseEntity<>(totalDeLivros, HttpStatus.OK);
    }

}
```

A consulta realizada no novo endpoint que criamos não faz sentido ser utilizada por um usuário em específico. Por que um usuário precisaria consultar o total de livros cadastrados na aplicação `bookserver`? Já a aplicação administrativa que disponibilizei no diretório `livro-spring-oauth2/capitulo-11` possui como requisito realizar tal consulta.

Antes de entrar em detalhes sobre a aplicação administrativa, precisamos configurar o grant type **Client Credentials**. Quando fizemos a configuração dos outros grant types, bastou adicionar o novo tipo usando o método `authorizedGrantTypes`. Contudo, além de adicionarmos o suporte ao novo grant type, vamos também configurar os dados da aplicação administrativa.

Por enquanto, continuaremos realizando as configurações do *client details* em memória. Abra a classe `ConfiguracaoOAuth2` e adicione o

trecho de código que configura o Client `cliente-admin` dentro método `configure` da classe interna `OAuth2AuthorizationServer`.

```
clients.inMemory()
    .withClient("cliente-app")
    .secret("123456")
    .authorizedGrantTypes("password", "authorization_code")
    .scopes("read", "write")
    .resourceIds(RESOURCE_ID)
.and()
    .withClient("cliente-admin")
    .secret("123abc")
    .authorizedGrantTypes("client_credentials")
    .scopes("read")
    .resourceIds(RESOURCE_ID);
```

Inicie a aplicação e prepare-se para realizar o primeiro teste interagindo com o endpoint de consulta administrativa. Faremos essa interação através da linha de comando, ou usando outra ferramenta como o Postman, apresentado anteriormente. Entretanto, antes de usar a API, vamos primeiro entender como solicitar o token de acesso junto ao Authorization Server.

11.2 Solicitação do token de acesso

A forma de se obter o token de acesso utilizando o grant type **Client Credentials** é uma das mais simples dentre os fluxos do OAuth 2.0. Precisamos enviar apenas o parâmetro `grant_type` com o valor `client_credentials` e, opcionalmente, o parâmetro `scope`. Esses parâmetros devem ser enviados como dados de formulário codificados com o formato `application/x-www-form-urlencoded`, e a requisição deve ser feita para o endpoint de solicitação de token `/oauth/token` usando o método `POST`.

Além dos parâmetros, é preciso identificar qual o Client que está solicitando o token de acesso, e isso é realizado através de

autenticação. No nosso caso, estamos usando *HTTP Basic Authentication*, portanto precisaremos enviar o `client_id` e o `client_secret` separados por `:` e codificados com Base 64. Lembre-se de que o Authorization Server é obrigado a autenticar o Client durante a solicitação de tokens.

Veja a seguir como pode ser feita uma solicitação de token de acesso através do comando `cURL`.

```
curl -X POST -H "Content-Type: application/x-www-form-urlencoded" \
-H "Authorization: Basic Y2xpZW50ZS1hZG1pbjoxMjNhYmM=" \
-d 'grant_type=client_credentials' \
"http://localhost:8080/oauth/token"
```

Ao executar esse comando, você deve receber uma resposta semelhante à mostrada a seguir (e que continua sendo bem parecida com as respostas obtidas ao usar outros *grant types*).

```
{
  "access_token": "1b4ea189-21a4-4c7d-b2e6-5eb016b50746",
  "token_type": "bearer",
  "expires_in": 43199,
  "scope": "read"
}
```

Apesar da semelhança na resposta, quando se utiliza o *grant type* Client Credentials, a especificação diz que o Authorization Server **não** deveria gerar um `refresh_token`, pois não existe a preocupação com experiência de usuário. No caso de comunicação *back-end* entre aplicações, o próprio Client é capaz de solicitar um token novo quando o atual estiver expirado.

Atenção, pois o fato de se utilizar tokens OAuth 2.0 que possuem tempo de validade acaba sendo uma vantagem em relação a se usar apenas *HTTP Basic Authentication* para permitir o acesso de aplicações internas às APIs. Além do mais, caso a aplicação disponibilize uma API protegida por OAuth 2.0 que é acessada por terceiros, essa mesma API pode ser acessada por aplicações internas que se aproveitam de uma estrutura já existente.

Como a aplicação já está funcionando, envie uma requisição para consultar o total de livros cadastrados na base de dados da aplicação `bookserver` utilizando o token de acesso obtido.

```
curl -X GET -H "Authorization: Bearer 1b4ea189-21a4-4c7d-b2e6-  
5eb016b50746" \  
"http://localhost:8080/api/v2/admin/total_livros"
```

A resposta para essa requisição deve ser apenas um número, que significa justamente a quantidade de livros cadastrados no sistema `bookserver`.

11.3 Implementação do Client

No diretório `livro-spring-oauth2/capitulo-11`, adicionei também um novo projeto chamado `admin`. Esse vai ser o exemplo de Client que usaremos para testar o grant type **Client Credentials**. Essa é uma aplicação bem simples que fornece um painel administrativo fictício para monitoração das aplicações dentro da empresa. Através dela, podemos acompanhar se o sistema está no ar e também podemos ver a quantidade de livros já cadastrados em `bookserver`.

Importe a aplicação `admin` para sua IDE e inicie o servidor que ficará disponível através da URL <http://localhost:9000>. Ao acessar essa URL, você deverá ver a seguinte tela:

Dashboard de administração

Sistema de livros (bookserver)

Status da aplicação: OK

Quantidade de livros cadastrados até agora: 0

Sistema de revistas (fictício)

Status da aplicação: OK

Quantidade de revistas publicadas: 150

Figura 11.2: Tela do painel administrativo

Repare que o valor ainda está zero onde devemos mostrar a quantidade de livros cadastrados na aplicação `bookserver`. Vamos começar a implementar o serviço que acessa o endpoint administrativo que criamos na aplicação `bookserver`.

Assim como fizemos para os outros *grant types*, encapsularemos a chamada para a API da aplicação `bookserver` através de um `service`. Abra a classe `BookserverService` da aplicação `admin`, e adicione o código a seguir dentro do método `getQuantidadeDeLivrosCadastrados`.

```
public long getQuantidadeDeLivrosCadastrados() {
    String token = clientCredentialsTokenService
        .getToken().getAccessToken();

    MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
    headers.add("Authorization", "Bearer " + token);

    String endpoint = "http://localhost:8080/api/v2/admin/total_livros";

    RequestEntity<Object> request = new RequestEntity<>(
        headers, HttpMethod.GET,
        URI.create(endpoint)
    );

    return sendRequest(request);
}
```

O método `getQuantidadeDeLivrosCadastrados` faz uma chamada simples para o endpoint `/api/v2/admin/total_livros`, passando apenas o atributo `Authorization` no header da requisição com um token obtido através do serviço `clientCredentialsTokenService`. Mas onde vamos definir os parâmetros para a requisição do token de acesso?

Para isso, precisamos alterar o método privado `getBody` da classe `ClientCredentialsTokenService`. É nesse método que vamos configurar os parâmetros exigidos pelo grant type **Client Credentials**.

```
private MultiValueMap<String, String> getBody() {
    MultiValueMap<String, String> dadosFormulario = new
LinkedMultiValueMap<>();

    dadosFormulario.add("grant_type", "client_credentials");
    dadosFormulario.add("scope", "read");

    return dadosFormulario;
}
```

Conforme já vimos anteriormente ao realizar alguns testes com o comando `cURL`, o grant type **Client Credentials** precisa receber apenas o parâmetro `grant_type` definido como `client_credentials`. Entretanto, além desse parâmetro, também já estamos passando o parâmetro `scope` para ilustrar o seu uso.

Não se esqueça que, para obter um token de acesso, quando usamos um `client` do tipo **confidential** é necessário se **autenticar** no Authorization Server. Para realizar a autenticação do Client `admin`, estamos enviando o `client_id` e o `client_secret` com os valores `cliente-admin` e `123abc`, respectivamente.

```
BasicAuthentication clientAuthentication = new
BasicAuthentication("cliente-admin", "123abc");
```

Assim como fizemos para os grant types **Resource Owner**, **Password Credentials** e **Authorization Code**, as credenciais do Client também são enviadas via *HTTP Basic Authentication*,

conforme mostrado na implementação do método `getHeader`, usado dentro da classe `ClientCredentialsTokenService`.

```
private HttpHeaders getHeader(BasicAuthentication clientAuthentication) {  
    HttpHeaders httpHeaders = new HttpHeaders();  
  
    httpHeaders.setContentType(  
        MediaType.APPLICATION_FORM_URLENCODED);  
    httpHeaders.setAccept(  
        Arrays.asList(MediaType.APPLICATION_JSON));  
    httpHeaders.add("Authorization", "Basic "  
        + clientAuthentication.getCredenciaisBase64());  
  
    return httpHeaders;  
}
```

Com todas as configurações realizadas, já podemos fazer um teste da aplicação para ver se tudo funciona conforme o esperado. Partindo do princípio que existam livros cadastrados na base de dados da aplicação `bookserver`, a tela de administração deve agora mostrar a quantidade de livros corretamente.

Inicie as aplicações `bookserver` e `admin`, acesse o link <http://localhost:9000> novamente e verifique se a quantidade de livros agora está correta.

Você reparou que toda vez que estamos buscando a quantidade de livros cadastrados na aplicação `bookserver`, o método `getQuantidadeDeLivrosCadastrados` da classe `BookserverService` está solicitando um token para o Authorization Server? Estamos usando esse modelo somente para facilitar a escrita do teste. Entretanto, em um sistema que vai ser colocado em produção, o ideal é que o token recuperado seja armazenado em algum local seguro (por exemplo, um banco de dados), para que não seja necessário solicitar um token de acesso toda vez que for acessar uma API.

11.4 Quando usar o grant type Client Credentials

O cenário ideal para se usar o *grant type Client Credentials* é fácil de ser identificado. Quando uma aplicação precisa acessar recursos em benefício próprio em vez de realizar operações em nome de um usuário, podemos usar **Client Credentials**.

Imagine nesse caso aplicações que precisam realizar comunicação através do *back-end*, dispensando completamente o uso de um *browser*. Além disso, como esse fluxo não conta com a presença de um usuário, a etapa de autorização já não faz mais sentido. Perceba também que estamos falando agora de Clients que sejam do tipo **confidencial**.

Se por um acaso você cogitar usar **Client Credentials** em aplicativos mobile para obter token de acesso, como você vai proteger as credenciais do Client dentro do aplicativo? Lembre-se de que armazenar as credenciais no próprio aparelho não é uma boa ideia. Como um Client mobile se trata de uma aplicação com perfil **público**, não se deve usar Client Credentials nesse caso.

Atualmente, muitos exemplos de aplicações distribuídas têm surgido seguindo a tendência de uma arquitetura baseada em **microservices**. Nesse tipo de arquitetura, como muitos serviços são disponibilizados através de APIs, é possível que alguns endpoints precisem ser distribuídos de forma protegida em sub-redes privadas. Isso cria camadas de segurança além da fronteira de aplicações que disponibilizam APIs públicas, conforme mostrado na figura a seguir.

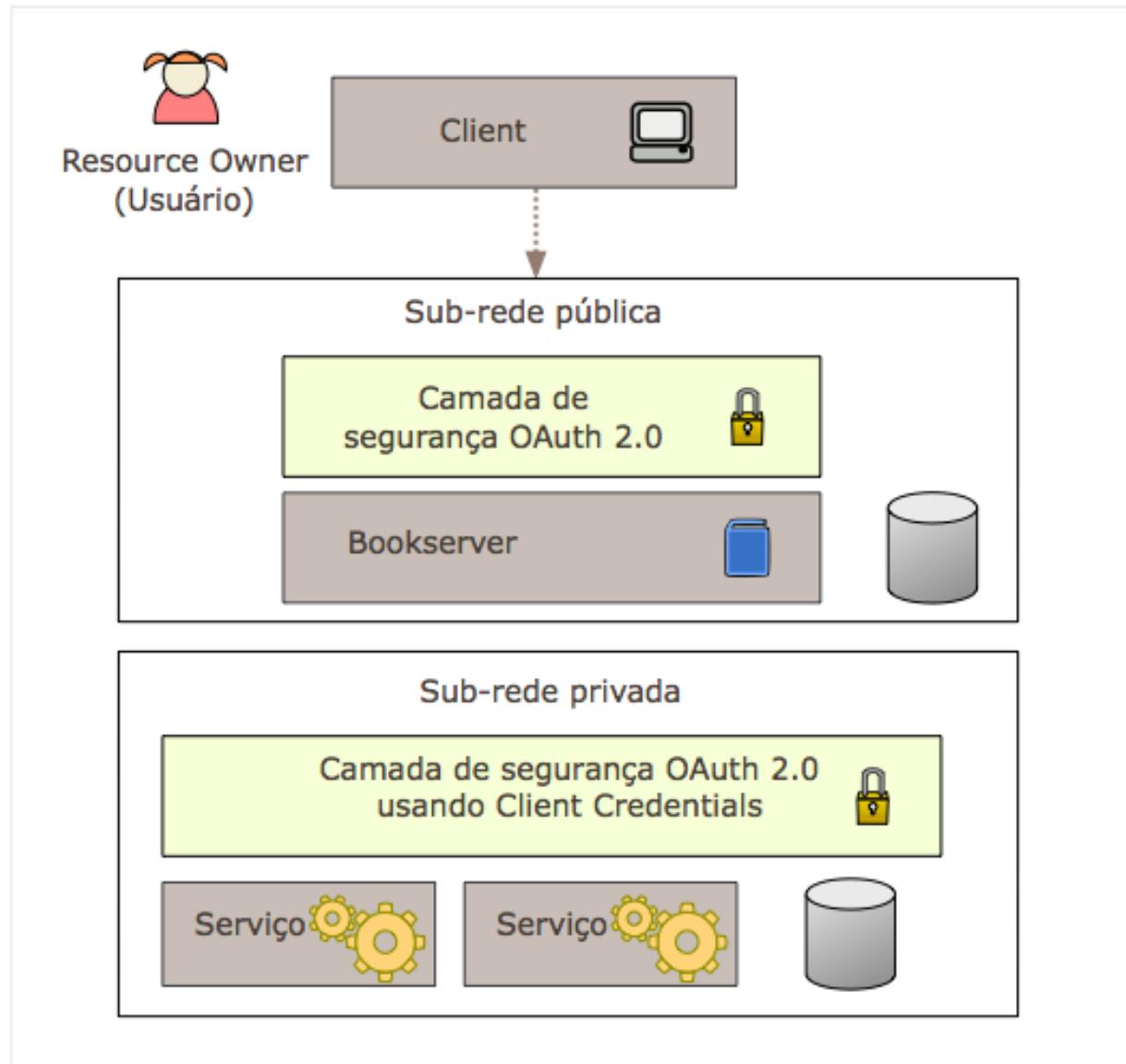


Figura 11.3: Camada adicional de segurança

Na imagem anterior, a aplicação `bookserver` acessa APIs de serviços distribuídos em uma sub-rede privada que possui regras baseadas em endereço IP e portas de acesso. Além das regras de roteamento, podemos proteger as APIs desses serviços com OAuth 2.0, assim como fizemos na aplicação `bookserver`.

Dessa forma, caso a aplicação `bookserver` seja comprometida devido a um ataque, será preciso quebrar a segurança adicionada na rede privada. Lembre-se de que esse cenário se trata apenas de

um exemplo para ilustrar como aplicações que pertencem a um mesmo domínio podem interagir entre si através de suas APIs protegidas por OAuth 2.0.

Diante desse cenário, uma questão importante costuma ser levantada. Por que não usar simplesmente *HTTP Basic Authentication* em vez de OAuth 2.0? A melhor resposta para essa pergunta depende do cenário em que estiver trabalhando.

Se você possui um ecossistema de APIs que já utiliza OAuth 2.0, é muito provável que você tenha Resource Servers que estão preparados para validar um token de acesso. Se sua aplicação já conta com Resource Servers interceptando as chamadas para sua API, por que usar *HTTP Basic Authentication*? Nesse caso, você pode aproveitar a arquitetura existente e se beneficiar com o mecanismo de expiração de token (trazendo mais proteção para sua API, mesmo que em uma escala pequena).

Agora caso você estiver usando **Client Credentials** sem expiração de token e, além disso, seu sistema nem utiliza OAuth 2.0, então é melhor repensar a arquitetura. Nesse caso, é melhor usar *HTTP Basic Authentication* e evitar a complexidade adicional do OAuth 2.0.

11.5 Conclusão

Neste capítulo, apresentei um *grant type* útil para ser usado em cenários em que o Client **não** acessa recursos de um Resource Owner. Mostrei como obter um token de acesso através de chamadas `CURL` e através da implementação de um Client REST.

O *grant type* **Client Credentials** é um dos mais simples de se utilizar, pois conta apenas com a fase de solicitação do token de acesso. Uma questão importante que se deve ter em mente é saber identificar quando usar esse *grant type*, evitando complexidades

desnecessárias. No próximo capítulo, vamos conhecer o grant type **Refresh Token** que é complementar aos *grant types* apresentados até agora.

CAPÍTULO 12

Refresh Tokens e escopos para aumentar a segurança

Durante a apresentação dos *grant types Authorization Code, Implicit, Resource Owner Password Credentials e Client Credentials*, realizamos as configurações para o Resource Server e Authorization Server de forma mais simples possível. Como exemplo dessa simplicidade, não estamos permitindo que o Client utilize *refresh tokens* para obter novos access tokens quando o que estiver sendo usado expirar.

Ademais, os tokens também estão sendo gerados com um tempo de expiração padrão definido pelo Spring Security OAuth2, que é de 43.200 segundos. Em algumas situações, como por exemplo quando se utiliza o grant type **Implicit**, é muito importante que se mantenha um tempo de expiração menor.

Outra questão importante que também não está sendo considerada nas aplicações de exemplo é a **validação do escopo** durante o acesso às APIs. Apesar de solicitarmos a geração dos tokens com determinados escopos, precisamos validar se o Client está acessando somente os recursos que tem direito. O Spring Security OAuth2 já fornece meios para configurarmos o tempo de expiração de tokens, o uso de refresh tokens e a validação de escopos. Neste capítulo, vamos explorar como funcionam tais configurações e quando podemos ou não fazer uso delas.

12.1 Quando usar refresh tokens e como configurar

Imagine que estamos usando o grant type **Authorization Code** e um usuário já permitiu o uso de seus recursos por parte de uma determinada aplicação terceira. Sem o uso de refresh tokens, quando o token de acesso expirar, o Client precisará redirecionar o usuário novamente para o Authorization Server, fazendo com que ele tenha de se autenticar novamente e fornecer as devidas permissões.

A experiência do usuário fica comprometida ao trabalharmos dessa maneira. Quando adicionamos suporte à geração de refresh token, damos aos Clients a possibilidade de solicitar um novo token de acesso sem precisar pedir autorização para o Resource Owner novamente. Entretanto, nem sempre faz sentido usar refresh tokens dependendo do grant type suportado.

Dentre os grant types em que o uso de refresh tokens faz sentido, podemos citar os grant types **Resource Owner Password Credentials** e **Authorization Code**. Para o grant type **Implicit**, como se trata de uma solução para aplicações que executam diretamente no *browser*, espera-se que o Resource Owner esteja sempre presente para autorizar o Client novamente. Além disso, o Authorization Server pode fornecer mecanismos de sessão para que o Resource Owner não precise se autenticar e autorizar um Client que já foi autorizado anteriormente (a menos que o Client esteja solicitando novos escopos).

O uso de refresh tokens para o grant type **Client Credentials** faz menos sentido ainda, pois a fase de autorização não existe, logo não existe a preocupação com usabilidade do Resource Owner. Além disso, espera-se que o Client seja capaz de solicitar um novo token usando o fluxo padrão. Afinal, trata-se apenas de uma requisição para o endpoint `/oauth/token` enviando dados que o próprio Client já possui (como `client_id` e `client_secret`).

Considerando os grant types **Resource Owner Password Credentials** e **Authorization Code**, vamos adicionar o suporte ao

`refresh_token` na aplicação `bookserver`, que está disponível no diretório `livro-spring-oauth2/capitulo-12`.

12.2 Adicionando suporte a refresh token

Importe o projeto `bookserver` para sua IDE e abra a classe `ConfiguracaoOAuth.OAuth2AuthorizationServer`. Vamos adicionar o suporte ao refresh token para o Client que está configurado com os grant types **Authorization Code** e **Resource Owner Password Credentials**, ou seja, o Client cujo `client_id` é `cliente-app`.

Além de adicionar o suporte ao refresh token, também vamos configurar um tempo de expiração menor para o token de acesso para facilitar os testes com refresh token.

```
@Override
public void configure(ClientDetailsServiceConfigurer clients)
    throws Exception {
    clients.inMemory()
        .withClient("cliente-app")
        .secret("123456")
        .scopes("read", "write")
        .resourceIds(RESOURCE_ID)
        .authorizedGrantTypes(
            "password", "authorization_code",
            "refresh_token")
        .accessTokenValiditySeconds(120);
    // outras configurações omitidas
}
```

Observe que o suporte aos **refresh tokens** é adicionado como um grant type, pois se trata de um fluxo para obtenção de token de acesso, assim como os outros grant types. Não esqueça de adicionar a configuração que define o tempo de expiração usando o método `accessTokenValiditySeconds` com 120 segundos.

Inicie a aplicação `bookserver` e tente obter um token de acesso usando o grant type **Authorization Code** ou **Resource Owner Password Credentials**. Caso não se lembre de como obter um token de acesso, vale a pena fazer uma revisão dos capítulos anteriores nos quais vimos em detalhes cada um desses grant types. No momento em que você solicitar um token de acesso através do endpoint `/oauth/token`, a resposta do servidor será um pouco diferente da que recebíamos anteriormente.

```
{  
  "access_token": "5f64730a-2906-4695-80bc-d73b6171685e",  
  "token_type": "bearer",  
  "refresh_token": "313abf0f-fbd2-4958-8dfd-992cbbefca8f",  
  "expires_in": 119,  
  "scope": "read write"  
}
```

A nova resposta contém mais um atributo chamado `refresh_token` que poderemos usar para obter novos tokens de acesso quando o atual expirar. Ao tentar usar o token de acesso após dois minutos para realizar qualquer operação através da API da aplicação `bookserver`, a resposta devolvida pelo servidor será semelhante ao JSON a seguir.

```
{  
  "error": "invalid_token",  
  "error_description": "Access token expired: 5f64730a-2906-4695-80bc-  
d73b6171685e"  
}
```

E agora, como podemos obter um novo token de acesso? Para obter um novo token, precisamos usar o **refresh token** recebido no momento da solicitação do token de acesso e enviar para o servidor através de uma requisição `HTTP POST` para o endpoint `/oauth/token`, conforme mostrado no comando `CURL` adiante.

Para realizar a solicitação do novo token, o Client precisa se autenticar passando o Client ID e o Client Secret via *HTTP Basic Authentication*. Ele também precisa informar o `grant_type` e o

`refresh_token` como atributos de formulário codificados com `x-www-form-urlencoded`.

```
curl -X POST --user cliente-app:123456 \
-d 'grant_type=refresh_token&refresh_token=313abf0f-fbd2-4958-8dfd-992cbbefca8f&scope=read write' \
"http://localhost:8080/oauth/token"
```

Porém, ao executar o comando anterior, o servidor devolve um erro informando que é necessário um `UserDetailsService`.

```
{
  "error": "server_error",
  "error_description": "UserDetailsService is required."
}
```

Acontece que, internamente, quando usamos o grant type **Resource Owner Password Credentials**, o Spring Security OAuth2 recupera informações do usuário que forneceu a autorização que deu origem ao token e refresh token que estão sendo usados, e tenta reautenticar o usuário. Por que o Spring Security OAuth2 faz isso?

A questão é que o Resource Owner pode, por algum motivo, ter alterado a própria senha, o que deve **invalidar** a autorização dada pelo Resource Owner e, por consequência, o refresh token sendo usado. Para resolver esse problema, precisamos apenas informar para o `configurer AuthorizationServerEndpointsConfigurer` qual o `UserDetailsService` que deve ser usado para validar o Resource Owner. Vale lembrar que essa configuração deve ser feita na classe `ConfiguracaoOAuth2 OAuth2AuthorizationServer`.

```
@Autowired
private AuthenticationManager authenticationManager;

@Autowired
private UserDetailsService userDetailsService;

@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)
```

```

throws Exception {
    // @formatter:off
    endpoints
        .authenticationManager(authenticationManager)
        .userDetailsService(userDetailsService);
    // @formatter:on
}

```

Além de configurar o `authenticationManager`, agora também estamos configurando o `userDetailsService`, que é o objeto responsável por recuperar os usuários cadastrados no banco de dados da aplicação.

Inicie a aplicação `bookserver` novamente e utilize o fluxo adequado para obter um token de acesso. Em seguida, realize alguns testes acessando a API de consulta de livros pelo endpoint

`http://localhost:8080/api/v2/livros`.

Quando passarem os dois minutos de validade do token, o servidor novamente retornará um erro informando que o token expirou.

Quando isso acontecer, tente solicitar um novo token utilizando o `refresh_token` fornecido pelo Authorization Server, conforme mostrado a seguir. Não se esqueça de sempre usar os **tokens que você tiver recebido** durante a execução dos testes.

```

curl -X POST --user cliente-app:123456 \
-d 'grant_type=refresh_token&refresh_token=313abf0f-fbd2-4958-8dfd-
992cbbefca8f&scope=read write' \
"http://localhost:8080/oauth/token"

```

Ao solicitar um novo token de acesso através do grant type **Refresh Token**, a resposta obtida deve ser semelhante à mostrada a seguir:

```

{
    "access_token": "d167016d-81ca-4ea1-8572-2b0f8f7722fc",
    "token_type": "bearer",
    "refresh_token": "313abf0f-fbd2-4958-8dfd-992cbbefca8f",
    "expires_in": 119,
    "scope": "read write"
}

```

Analisando a resposta do servidor, podemos notar que o `refresh_token` recebido é igual ao `refresh_token` que usamos para obter um novo token. De acordo com a especificação, o Authorization Server **pode** gerar um novo `refresh_token` e, caso isso seja realizado, o Client deve descartar o `refresh_token` antigo. Além disso, o Authorization Server tem a opção de revogar ou não o `refresh_token` antigo.

Ao configurar um tempo de validade para os tokens de acesso, trazemos mais segurança para a aplicação que, em conjunto com o uso de refresh tokens, permite que a experiência do usuário não seja comprometida. Como nesse caso o processo de solicitação do novo token é muito similar ao de solicitação de token normal, não estamos mostrando um exemplo de Client. Mas como exercício, você pode importar o projeto Client que usamos na apresentação dos grant types Authorization Code ou Resource Owner Password Credentials e implementar o uso de refresh tokens.

12.3 Validando os escopos durante o acesso a uma API

Quando redirecionamos o Resource Owner para o Authorization Server, já sabemos que, além de se autenticar o usuário, precisamos autorizar alguns dos escopos solicitados pelo Client. Caso contrário, o Authorization Server devolve um erro de permissão negada. Existem situações em que queremos que um token só possa ser usado para realizar operações que foram permitidas pelo usuário.

Se você estiver implementando um **OAuth Provider** do zero, você pode criar seus próprios mecanismos de validação de escopo dos tokens. Entretanto, o Spring Security OAuth2 se beneficia do mecanismo de controle de acesso, baseado em expressões, fornecido pelo **Spring Security**.

Tais expressões podem ser usadas para validar o acesso a um método, ou a um endpoint definido em um `controller`. Como exemplos de expressões, temos `hasRole`, `hasAnyRole`, `hasAuthority` e `hasAnyAuthority`. Contudo o **Spring Security OAuth2** fornece expressões adicionais conforme as mostradas na tabela a seguir.

Expressão	Descrição
<code>hasScope(string)</code>	Permite validar se o token usado na requisição possui o escopo informado na expressão.
<code>hasAnyScope(string...)</code>	Verifica se o token usado possui um dos escopos informados.
<code>denyOAuthClient</code>	Bloqueia o acesso para requisições OAuth.
<code>isOAuth()</code>	Permite acesso somente através de requisições OAuth.
<code>clientHasRole(string)</code>	Valida se o Client possui a <code>role</code> informada.
<code>clientHasAnyRole(string...)</code>	Valida se o Client possui uma das <code>roles</code> informadas.

Na tabela anterior, apresentei algumas expressões mais usadas. Porém, existem outras que podem ser consultadas na documentação, disponível em <http://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/provider/expression/OAuth2SecurityExpressionMethods.html>.

12.4 Configuração dos endpoints com escopos

Antes de configurar qual escopo é necessário para acessar um endpoint, precisamos habilitar as anotações que permitem utilizar as expressões para validação de controle de acesso. As anotações que podemos usar com as expressões mostradas anteriormente são `@PreAuthorize`, `@PreFilter`, `@PostAuthorize` e `@PostFilter`. Então, como habilitamos o uso dessas anotações?

Abra a classe `BookServerApplication` do projeto `bookserver`, e adicione a anotação `@EnableGlobalMethodSecurity` no nível da classe, conforme mostrado a seguir:

```
@SpringBootApplication
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class BookServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(BookServerApplication.class, args);
    }
}
```

Através da anotação `@EnableGlobalMethodSecurity`, habilitamos o uso de validações pré e pós-execução pelo atributo `prePostEnabled`. Agora tudo o que precisamos fazer é informar qual é o escopo permitido para acessar algum endpoint da aplicação `bookserver`. Por enquanto, temos apenas um endpoint de consulta, então vamos permitir o acesso apenas a tokens com escopo de leitura.

```
@RestController
@RequestMapping({"/api/livros", "/api/v2/livros"})
public class LivrosApiController {

    @PreAuthorize(value = "#oauth2.hasScope('read')")
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<?> livros() {
    }
}
```

Adicionamos a validação através da anotação `@PreAuthorize`, na qual configuramos a expressão `#oauth2.hasScope('read')` para garantir que

apenas tokens com escopo de leitura possam ser usados para consultar livros de um Resource Owner.

Para simular um cenário de falha, vamos criar um token de acesso que possua apenas o escopo de escrita. Primeiro, reinicie a aplicação `bookserver` e siga para a URL a seguir:

```
http://localhost:8080/oauth/authorize?client_id=cliente-
app&response_type=code&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallba
ck&scope=read+write&state=teste
```

Autentique-se com o login de um **Resource Owner**, ou cadastre-se através da interface web mostrada a seguir.



Login

Login:

Senha:

Enviar

Voltar

Não tem conta? [Faça seu cadastro](#)

Figura 12.1: Tela de login na aplicação bookserver

Após realizar o login, aprove apenas o acesso ao escopo `write`.

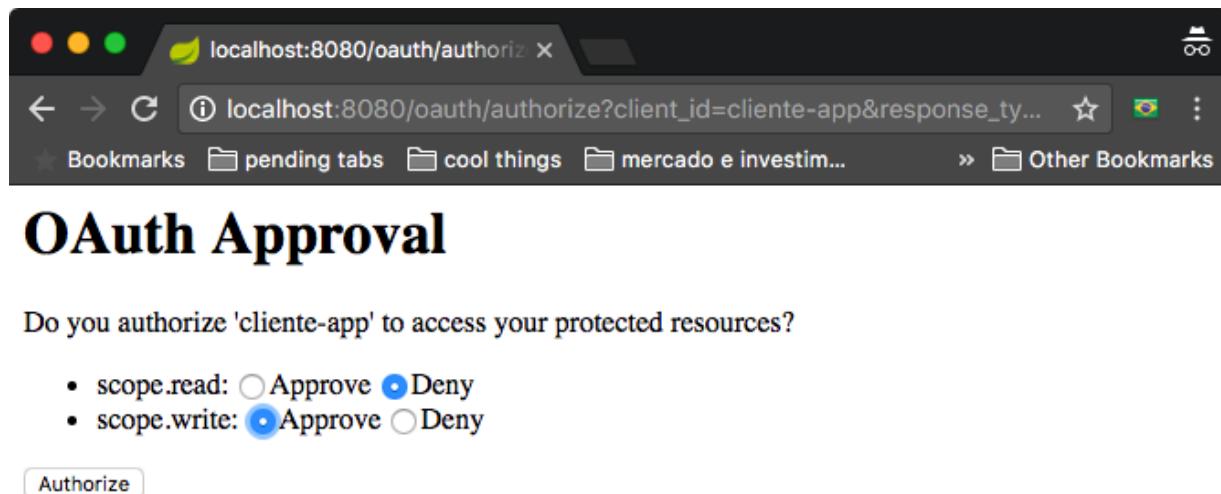


Figura 12.2: Tela de autorização com escopo write

Clique em autorizar e aguarde o Authorization Server realizar o redirecionamento para o endpoint de *callback* da aplicação Client, com o `authorization_code` embutido na URL de resposta.

`http://localhost:9000/callback?code=qpKtTC&state=teste`

Copie o valor do parâmetro `code` retornado na URL de *callback*, e use-o como parâmetro da requisição de access token, conforme mostrado a seguir. No meu caso, o código retornado foi `qpKtTC`.

```
curl -X POST --user cliente-app:123456 \
-H "Content-Type: application/x-www-form-urlencoded" \
-d
'grant_type=authorization_code&redirect_uri=http://localhost:9000/callback
&scope=read write&code=qpKtTC' "http://localhost:8080/oauth/token"
```

Conforme você já deve esperar, será mostrado no seu console uma resposta semelhante à mensagem seguinte:

```
{
  "access_token": "fa8ceb23-02fd-40e4-b889-64f520417c53",
  "token_type": "bearer",
  "refresh_token": "c1b730e5-90ba-4c32-a382-26246b1a0f1d",
  "expires_in": 119,
```

```
        "scope": "write"
    }
```

Agora tente acessar o endpoint de consulta de livros usando o token obtido.

```
curl -X GET -H "Authorization: Bearer fa8ceb23-02fd-40e4-b889-  
64f520417c53" \  
"http://localhost:8080/api/v2/livros"
```

Como o **Resource Owner** não forneceu permissão para o escopo `read` durante a geração do token de acesso, então o acesso deve ser negado, de forma que a aplicação `bookserver` retornará a seguinte resposta:

```
{
  "error": "access_denied",
  "error_description": "Access is denied"
}
```

12.5 Configurando roles para os usuários

Além do mecanismo de validação do escopo que usamos no exemplo anterior, também podemos realizar o controle de acesso à API de acordo com *roles* definidas para o **Resource Owner**.

ROLES NO SPRING SECURITY

As *roles* representam basicamente o papel de um usuário no sistema. Como exemplo de *role*, podemos ter uma `ROLE_ADMIN` que classifica usuários com acesso de administrador. Elas podem ser usadas para agrupar usuários que possuem uma característica em comum, de acordo com o que ele pode ou não fazer no sistema.

As *roles* são apenas `strings` com um valor arbitrário. Entretanto, quando queremos obter quais são as *roles* de um usuário autenticado, usamos um método da interface `UserDetails`, chamado `getAuthorities`. Este, por sua vez, retorna uma coleção de objetos do tipo `GrantedAuthority`.

Para a aplicação `bookserver`, vamos imaginar uma situação na qual podemos ter dois tipos de usuário na aplicação: funcionários e usuários comuns. Teremos esses tipos de usuário definidos através das *roles* `ROLE_FUNCIONARIO` e `ROLE_USUARIO_COMUM`, respectivamente.

A ideia aqui é que as APIs de controle de livros só possam permitir que usuários comuns deleguem acesso aos seus recursos. Caso um Client tente usar um token autorizado por um funcionário, a aplicação não poderá permitir o acesso aos recursos de um funcionário. A razão disso pode ser uma questão de segurança, mas não vamos nos preocupar com a aplicabilidade dessa regra de negócio, pois o foco aqui é entender o uso das *roles*.

Para adicionar essa regra na aplicação, podemos simplesmente alterar o método `livros` da classe `LivrosApiController` para considerar mais uma validação através da expressão `hasRole`.

```
@PreAuthorize(  
    value = "#oauth2.hasScope('read') and hasRole('ROLE_USUARIO_COMUM')")  
@RequestMapping(method = RequestMethod.GET)  
public ResponseEntity<?> livros() {
```

```
// código omitido  
}
```

Inicie a aplicação `bookserver` e tente gerar um token de acesso usando o grant type Authorization Code. Na tela onde o Resource Owner autoriza os escopos, escolha todas opções (`read` e `write`). Ao tentar usar o token para acessar o endpoint <http://localhost:8080/api/v2/livros>, você deve receber uma resposta com status `http 403`, mesmo com o Resource Owner fornecendo acesso a todos os escopos.

```
{  
    "error": "access_denied",  
    "error_description": "Access is denied"  
}
```

Por que isso acontece? Isso acontece pois os usuários da nossa aplicação não possuem `roles` configuradas. Vamos adicionar uma configuração de `role` fixa para nossos usuários através do método `getAuthorities` na classe `ResourceOwner`. Em uma aplicação real, as `roles` poderiam ser recuperadas do banco de dados.

```
public class ResourceOwner implements UserDetails {  
    // código omitido  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        List<SimpleGrantedAuthority> roles = new ArrayList<>();  
        roles.add(new SimpleGrantedAuthority("ROLE_USUARIO_COMUM"));  
        return roles;  
    }  
}
```

Reinic peace a aplicação, gere um novo token de acesso e tente novamente acessar o endpoint de consulta de livros usando o novo token. Agora como esse token foi gerado para um usuário que contém a `role` `ROLE_USUARIO_COMUM`, o acesso é permitido e obtemos uma resposta válida contendo os livros do Resource Owner.

12.6 Relacionando escopos e roles

Agora que você conhece bem o uso das *roles* e dos escopos, podemos explorar uma configuração bem interessante. Não seria legal se os escopos que o Resource Owner pudesse autorizar fossem apenas os que casassem com as *roles* que ele possui? Dessa forma, um Resource Owner não correria o risco de fornecer permissões para um Client que fossem além das próprias permissões que ele tivesse.

Conforme explicado por Dave Syer (líder do projeto Spring Security OAuth2) em um post no Stackoverflow, as *roles* e os escopos são strings arbitrárias que podem ser tratadas como a mesma coisa. Caso esteja curioso para ler mais sobre isso, acesse <http://stackoverflow.com/questions/22417780/using-scopes-as-roles-in-spring-security-oauth2-provider>.

Para adicionar essa funcionalidade em nosso projeto `bookserver`, é preciso que o bean do tipo `AuthorizationEndpoint` (classe do próprio Spring Security OAuth2) seja capaz de **verificar** os escopos do Resource Owner na **etapa de autorização**. Para isso, altere o seguinte método de configuração da classe `OAuth2AuthorizationServer` para podermos customizar o objeto responsável por criar uma requisição de autorização. O objeto que estamos falando é definido através da classe `DefaultOAuth2RequestFactory`.

```
@EnableAuthorizationServer
protected static class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {

    // outros atributos omitidos

    @Autowired
    private ClientDetailsService clientDetailsService;

    @Override
```

```

    public void configure(AuthorizationServerEndpointsConfigurer
endpoints) throws Exception {
    DefaultOAuth2RequestFactory requestFactory = new
DefaultOAuth2RequestFactory(clientDetailsService);
    requestFactory.setCheckUserScopes(true);

    // @formatter:off
    endpoints
        .userDetailsService(userDetailsService)
        .authenticationManager(authenticationManager)
        .requestFactory(requestFactory)
    ;
    // @formatter:on
}
}

```

Estamos customizando o objeto do tipo `DefaultOAuth2RequestFactory` para que, quando uma requisição de autorização for criada, os escopos do usuário sejam validados. Para que os escopos sejam validados, estamos definindo o atributo `checkUserScopes` como `true`.

Além disso, para criar uma instância de `DefaultOAuth2RequestFactory`, passamos como parâmetro uma instância de `ClientDetailsService`, que injetamos no começo da classe `OAuth2AuthorizationServer`. Agora, para que isso funcione, vamos adicionar o escopo `read` para os Resource Owners da aplicação:

```

public class ResourceOwner implements UserDetails {
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<SimpleGrantedAuthority> roles = new ArrayList<>();
        roles.add(new SimpleGrantedAuthority("ROLE_USUARIO_COMUM"));
        roles.add(new SimpleGrantedAuthority("read"));

        return roles;
    }
}

```

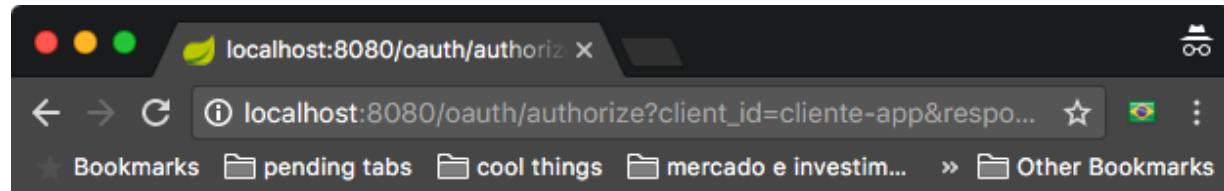
Na configuração anterior, estamos informando que o Resource Owner pode autorizar o escopo `read` para uma aplicação terceira

que faça integração via OAuth 2.0. Entretanto, precisamos configurar também as authorities que um Client possui. Isso vai determinar quais escopos baseados em roles o Client pode solicitar.

Vamos então adicionar os authorities que o Client pode solicitar. Altere o código da classe ConfiguracaoOAuth2AuthorizationServer dentro do método que usamos para configurar o *client details* do cliente-app .

```
clients.inMemory()
    .withClient("cliente-app")
    .secret("123456")
    .scopes("read", "write")
    .resourceIds(RESOURCE_ID)
    .authorities("read", "write")
    .authorizedGrantTypes(
        "password", "authorization_code")
```

Inicie a aplicação novamente. Ao utilizar o grant type Authorization Code, perceba que, com as configurações realizadas, o usuário só pode autorizar o escopo read .



OAuth Approval

Do you authorize 'cliente-app' to access your protected resources?

- scope.read: Approve Deny

[Authorize](#)

Figura 12.3: Tela de autorização mostrando como opção de escopo apenas o escopo que o Resource Owner também possui

Caso o Resource Owner autorizar o acesso ao escopo `read`, receberemos o `authorization_code` na URL que poderá ser usado para obter um token de acesso. Realize esse teste e observe que o token gerado terá apenas o escopo `read`.

```
{  
    "access_token": "7f0aa3dd-4ad5-4e8d-849c-fc21ebf3e91b",  
    "token_type": "bearer",  
    "refresh_token": "d59ef34d-d7b4-4109-a4a7-42053f0c6674",  
    "expires_in": 119,  
    "scope": "read"  
}
```

Se na etapa de autorização o Client solicitar um escopo que não esteja configurado, assim que o Resource Owner se autenticar, ele será redirecionado de volta para o Client com o seguinte parâmetro na URL de callback:

```
error=invalid_scope
```

12.7 Conclusão

Este capítulo apresentou algumas questões mais avançadas a respeito do protocolo OAuth 2.0 e, principalmente, do Spring Security OAuth2. Algumas vezes, os tipos de configurações que mostrei não são usados nos projetos em produção. É comum que validações de escopo e o uso de refresh tokens sejam deixados para depois.

Dependendo do projeto em que estiver trabalhando, talvez não seja um problema tão grande não utilizar tais configurações de início. Entretanto, espero que, ao finalizar este capítulo, você tenha entendido a importância do que foi estudado e tenha aumentado o conhecimento sobre o próprio Spring Security quando mostrei o uso das expressões usadas para controle de acesso.

Agora você já conhece bastante sobre o protocolo OAuth 2.0, mas ainda é importante conhecer outros recursos importantes sobre o Spring Security OAuth2. Um recurso importante que não pode faltar em um projeto que vai ser colocado em produção é usar um banco de dados de verdade para nosso **OAuth Provider**. Vamos para o capítulo seguinte, para dar um passo em direção a uma solução mais profissional.

CAPÍTULO 13

Adicionando suporte a banco de dados no projeto

Até agora, configuramos todos os dados relacionados ao Client em um banco de dados em memória, fornecido pelo Spring Security OAuth2 através da classe `InMemoryClientDetailsService`. Essa forma de armazenar os dados do Client não é indicada para ser usada em produção, pois toda vez que a aplicação for reiniciada, todos os dados serão perdidos.

Para resolver esse problema, precisamos de um banco de dados em que seja possível **persistir** os dados do Client. Neste capítulo, apresentarei como realizar essa configuração utilizando um banco de dados relacional que será acessado através da classe `JdbcClientDetailsService`, fornecida pelo Spring Security OAuth2. Ambas as classes `InMemoryClientDetailsService` e `JdbcClientDetailsService` implementam a interface `ClientDetailsService`, conforme mostra o diagrama de classes a seguir.

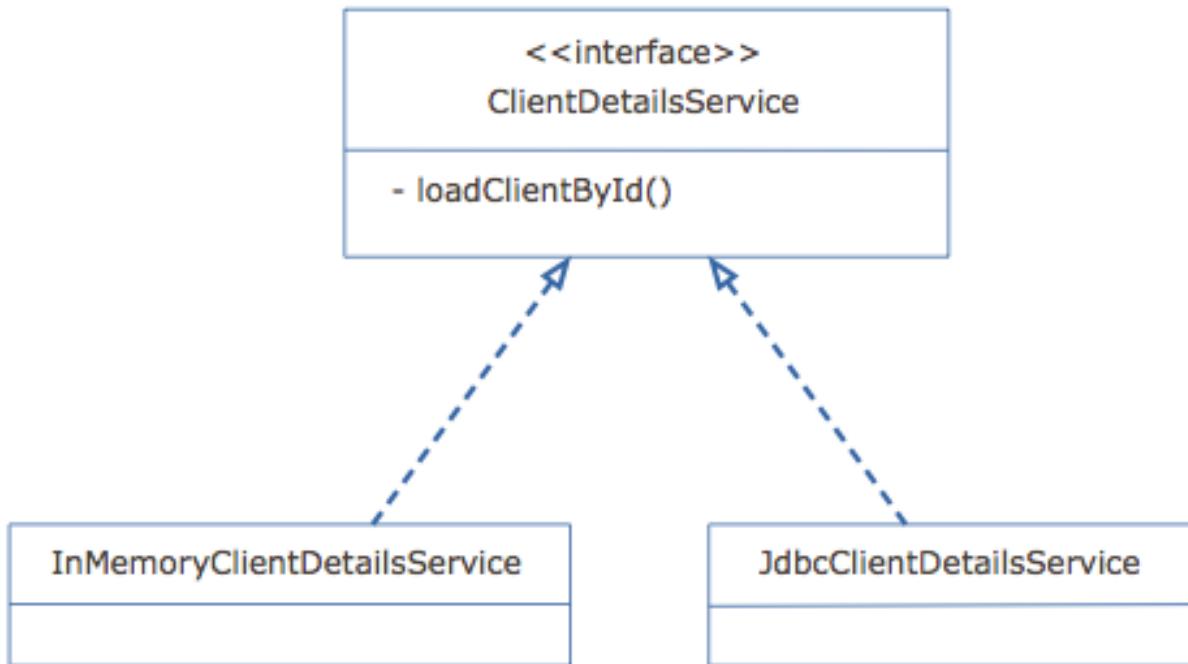


Figura 13.1: Hierarquia para gerenciamento de ClientDetails

Note que existe uma grande semelhança entre a interface `ClientDetailsService` do Spring Security OAuth2 com a interface `UserDetailsService` do Spring Security que usamos para configurar a autenticação do usuário final na aplicação `bookserver`. Assim como implementamos o método `loadUserByUsername` na classe `DadosDoUsuarioService`, as classes que implementam `ClientDetailsService` também fazem de maneira bem parecida através do método `loadClientById`.

A principal diferença entre esses dois métodos está no retorno. Enquanto a interface `UserDetailsService` é usada para obter um objeto que implemente `UserDetails`, a interface `ClientDetailsService` é usada para obter um objeto que implemente `ClientDetails`.

13.1 A definição do banco de dados

Para configurar a autenticação do usuário, criamos algumas tabelas no banco de dados usando uma estrutura que faz sentido para o domínio em que estamos trabalhando. Entretanto, qual deve ser a estrutura necessária para mantermos os dados do Client? Podemos identificar alguns dados que precisam ser mantidos simplesmente analisando a configuração em memória que temos até agora.

```
clients.inMemory()
    .withClient("cliente-app")
    .secret("123456")
    .scopes("read", "write")
    .resourceIds(RESOURCE_ID)
    .authorities("read", "write")
    .authorizedGrantTypes(
        "password",
        "authorization_code",
        "refresh_token")
    .accessTokenValiditySeconds(120)
```

Cada um dos atributos mostrados no trecho de código anterior pode ser definido como um campo em uma tabela na qual vamos manter os dados do Client. Embora seja possível fazer tudo isso manualmente, a classe `JdbcClientDetailsService` já espera por uma estrutura específica, bastando criar um modelo que atenda tal estrutura. Mas como podemos descobrir qual é essa estrutura?

Se você abrir a classe `JdbcClientDetailsService`, notará trechos de código SQL definidos através de constantes, conforme mostrado a seguir:

```
private static final String CLIENT_FIELDS_FOR_UPDATE = "resource_ids,
scope,
+
"authorized_grant_types, web_server_redirect_uri, authorities,
access_token_validity,
+
"refresh_token_validity, additional_information, autoapprove";

private static final String CLIENT_FIELDS = "client_secret, " +
CLIENT_FIELDS_FOR_UPDATE;

private static final String BASE_FIND_STATEMENT = "select client_id, " +
```

```

CLIENT_FIELDS + " from oauth_client_details";

private static final String DEFAULT_FIND_STATEMENT = BASE_FIND_STATEMENT +
" order by client_id";

private static final String DEFAULT_SELECT_STATEMENT = BASE_FIND_STATEMENT
+ " where client_id = ?";

// muito código omitido

```

A essa altura você deve estar pensando que é preciso *hackear* o código-fonte do Spring Security OAuth2 para descobrir como montar o banco de dados que contém os detalhes do Client, porém não se preocupe. Os desenvolvedores do Spring Security OAuth2 já deixaram no repositório do projeto um arquivo com todos os comandos SQL necessários para criar a estrutura de armazenamento dos detalhes do Client. Esse arquivo com os comandos SQL está disponível no GitHub, através do link <https://github.com/spring-projects/spring-security-oauth/blob/master/spring-security-oauth2/src/test/resources/schema.sql>.

Veja como deve ser a estrutura para os dados do Client:

```

create table oauth_client_details (
    client_id VARCHAR(256) PRIMARY KEY,
    resource_ids VARCHAR(256),
    client_secret VARCHAR(256),
    scope VARCHAR(256),
    authorized_grant_types VARCHAR(256),
    web_server_redirect_uri VARCHAR(256),
    authorities VARCHAR(256),
    access_token_validity INTEGER,
    refresh_token_validity INTEGER,
    additional_information VARCHAR(4096),
    autoapprove VARCHAR(256)
);

```

Você já deve estar acostumado com a maioria dos campos definidos para a tabela `oauth_client_details`, pois já usamos alguns na

configuração em memória. Mas a questão agora é: precisaremos criar código para manipular os dados dessa tabela assim como fizemos na classe `DadosDoUsuarioService` que implementa `UserDetailsService`?

Para alívio do desenvolvedor, o Spring Security OAuth2 já fornece todo o código de infraestrutura para manipular o banco de dados. Basta criar as tabelas definidas no arquivo `SQL` disponível no site do projeto Spring Security OAuth2, e injetar as classes corretas para que a integração com banco de dados seja feita de maneira bem simples.

13.2 Criando o banco de dados para o Authorization Server

Vamos colocar a mão na massa criando um banco de dados no MySQL para o Authorization Server que configuramos na aplicação `bookserver`. Os códigos de exemplo que usaremos agora estão no diretório `livro-spring-oauth2/capitulo-13`. Importe o projeto `bookserver`, abra o arquivo `database-auth-server.sql`, copie todo o conteúdo e execute no MySQL. Após executar esse `script SQL`, as seguintes tabelas devem ter sido criadas.

Tabela	Descrição
<code>oauth_client_details</code>	Contém os dados do Client como <code>client_id</code> , <code>client_secret</code> e <code>redirect_url</code> .

Tabela	Descrição
<code>oauth_access_token</code>	Contém os access tokens gerados para cada Client, sendo indicado que os tokens sejam armazenados de maneira criptografada.

Tabela	Descrição
<i>oauth_client_token</i>	Tabela que relaciona o Client com os tokens.
<i>oauth_refresh_token</i>	Armazena os dados relacionados aos refresh tokens gerados para o Client.
<i>oauth_code</i>	Mantém no banco de dados o <i>authorization_code</i> gerado a partir do <i>grant type Authorization Code</i> , para ser utilizado posteriormente na etapa de solicitação de token.
<i>oauth_approvals</i>	Mantém registro de todas as aprovações que o Resource Owner deu para cada Client. Para que essa tabela seja usada, é preciso configurar um <i>JdbcApprovalStore</i> .

13.3 Configurando a aplicação para usar o banco de dados

Agora precisamos configurar um `datasource` para que a aplicação possa se conectar com o banco de dados que criamos, porém nossa aplicação já possui uma definição de `datasource` no arquivo `application.yml`. Como vamos configurar mais de um `datasource` na mesma aplicação?

No Spring, podemos realizar essa configuração de diferentes maneiras, mas vamos partir para uma abordagem simples utilizando uma definição customizada. Abra o arquivo `application.yml` e observe que estamos usando duas estruturas para os `datasources`, sendo uma chamada `ds-bookserver` e a outra `ds-oauth`.

```

## múltiplas configurações de datasource do Spring
spring:
  ds-bookserver:
    url: jdbc:mysql://localhost/bookserver
    username: bookserver
    password: 123
    driver-class-name: com.mysql.jdbc.Driver
  ds-oauth:
    url: jdbc:mysql://localhost/bookserver_oauth
    username: bookserver_oauth
    password: 123
    driver-class-name: com.mysql.jdbc.Driver

```

Com configurações específicas para cada *data source*, precisamos também declarar *beans* correspondentes, sendo um para carregar o `DataSource` da aplicação `bookserver` e outro para carregar o `DataSource` do banco de dados OAuth2.

As declarações dos *beans* estão sendo definidas na classe `ConfiguracaoOAuthDB`, na qual as propriedades de `DataSource` que definimos no arquivo `application.yml` serão recuperadas de acordo com os prefixos correspondentes.

```

@Configuration
public class ConfiguracaoOAuthDB {

    @Bean(name = "dsOauth")
    @ConfigurationProperties(prefix = "spring.ds-oauth")
    public DataSource oauthDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Primary
    @ConfigurationProperties(prefix = "spring.ds-bookserver")
    public DataSource bookserverDataSource() {
        return DataSourceBuilder.create().build();
    }

}

```

Como estamos definindo dois objetos do tipo `DataSource` no contexto do Spring, o Spring Data JPA precisará diferenciar os `DataSources` para evitar conflitos na hora de injetar o `bean` nas classes internas do framework. Note que cada `DataSource` criado se trata de um `bean` que ficará disponível no contexto do Spring.

Para que o `DataSource` do banco de dados `bookserver` continue sendo injetado normalmente, vamos anotar o método `bookserverDataSource` com `@Primary`. Já o `DataSource` criado para o banco de dados OAuth2 poderá ser referenciado através de um qualificador, pois logo a seguir configuraremos o **Authorization Server** para usar o banco de dados que criamos nos passos anteriores, de maneira controlada. Ou seja, poderemos escolher manualmente qual `DataSource` nos interessa.

QUALIFICADORES NO SPRING

Quando usamos a anotação `@Autowired` para injetar um objeto de determinado tipo, o Spring buscará pelas possíveis implementações desse tipo que estejam disponíveis no contexto da aplicação. Caso mais de um `bean` do mesmo tipo seja carregado no contexto, o Spring precisa diferenciá-los.

Uma das maneiras de fazer essa diferenciação é através do uso de nomes para os beans. Através dos nomes, é possível usar a anotação `@Autowired` em conjunto com a anotação `@Qualifier("nomeDoBean")`. Os qualificadores ajudam o framework a identificar qual implementação específica de um `bean` deve ser injetada em uma classe.

13.4 Configuração do Authorization Server

Agora que as fontes de dados estão sendo carregadas corretamente, vamos finalmente configurar a utilização de banco de dados por parte do **Authorization Server**. Abra a classe `ConfiguracaoOAuth2.0Auth2AuthorizationServer` e apague todo o código existente, deixando essa classe da seguinte forma:

```
@EnableAuthorizationServer  
protected static class OAuth2AuthorizationServer  
    extends AuthorizationServerConfigurerAdapter {  
  
}
```

Para realizar as configurações, vamos injetar um `AuthenticationManager`, um `ClientDetailsService` e um `DataSource` qualificado como `ds0auth`. Atenção quando estiver importando a classe `DataSource`, pois esta deve ser a do pacote `javax.sql`.

```
@EnableAuthorizationServer  
protected static class OAuth2AuthorizationServer  
    extends AuthorizationServerConfigurerAdapter {  
  
    @Autowired  
    private AuthenticationManager authenticationManager;  
  
    @Autowired  
    private ClientDetailsService clientDetailsService;  
  
    @Autowired @Qualifier("ds0auth")  
    private DataSource dataSource;  
}
```

Lembre-se de que estamos injetando um `AuthenticationManager`, porque vamos dar suporte ao *grant type* Resource Owner Password Credentials. Caso não quiséssemos suportá-lo, não seria necessário configurar um `authenticationManager`.

Agora adicione os métodos a seguir na classe `ConfiguracaoOAuth2.0Auth2AuthorizationServer`, para definirmos os beans responsáveis por registrar os tokens e as aprovações do Resource Owner no banco de dados.

```
@Bean  
public TokenStore tokenStore() {  
    return new JdbcTokenStore(dataSource);  
}  
  
@Bean  
public ApprovalStore approvalStore() {  
    return new JdbcApprovalStore(dataSource);  
}
```

Para que os dados do Client, que agora estão armazenados no banco de dados, possam ser usados pela aplicação, adicione a seguinte configuração na classe `OAuth2AuthorizationServer`.

```
@Override  
public void configure(ClientDetailsServiceConfigurer clients)  
throws Exception {  
    clients.jdbc(dataSource);  
}
```

Agora, para que o Spring Security OAuth2 passe a usar os beans que declaramos anteriormente, adicione o seguinte método de configuração:

```
@Override  
public void configure(AuthorizationServerEndpointsConfigurer endpoints)  
throws Exception {  
    DefaultOAuth2RequestFactory requestFactory = new  
        DefaultOAuth2RequestFactory(clientDetailsService);  
    requestFactory.setCheckUserScopes(true);  
  
    endpoints  
        .authenticationManager(authenticationManager)  
        .requestFactory(requestFactory)  
        .approvalStore(approvalStore())  
        .tokenStore(tokenStore());  
}
```

Para que a aplicação continue funcionando com o **Client** cliente-app que estávamos usando até o momento, vamos fazer um `insert` manual na tabela `oauth_client_details` de acordo com o script

mostrado a seguir, que também se encontra no projeto `bookserver` dentro do arquivo `base-inicial.sql`.

```
INSERT INTO oauth_client_details
  (client_id, resource_ids, client_secret, scope,
  authorized_grant_types,
  web_server_redirect_uri, authorities, access_token_validity,
  refresh_token_validity,
  additional_information, autoapprove)
VALUES
  ('cliente-app', 'books', '123456', 'read,write',
  'password,authorization_code,refresh_token',
  'http://localhost:9000/callback', 'read,write', 3000, -1, NULL, 'false');
```

Antes de iniciar a aplicação, certifique-se de que as classes associadas à classe `Usuario` implementam `Serializable`, bem como a própria classe `Usuario`. Por enquanto, isso é necessário pois não definimos uma estratégia de criptografia para armazenar o token do usuário. Enquanto não adicionarmos essa estratégia, os dados do usuário e o token são gerados com base na serialização do objeto que representa o Resource Owner no banco de dados (no nosso caso, a classe `Usuario`).

Agora inicie a aplicação `bookserver` e faça um teste gerando um token para o Client `cliente-app` usando o grant type **Resource Owner Password Credentials**, conforme mostrado a seguir:

```
curl -X POST -H "Authorization: Basic Y2xpZW50ZS1hcHA6MTIzNDU2" \
-H "Content-Type: application/x-www-form-urlencoded" \
-H "Accept: application/json" \
-d
'grant_type=password&username=jujuba@mailinator.com&password=123&scope=rea
d write' \
"http://localhost:8080/oauth/token"
```

Assim que o Authorization Server retornar a resposta contendo o token de acesso, faça uma consulta na tabela `oauth_access_token` do banco de dados `bookserver-oauth`. Você poderá ver que um novo registro foi inserido.

Lembre-se de usar um usuário que você já tenha cadastrado na aplicação `bookserver`. Caso contrário, se o usuário que você estiver passando no parâmetro `username` não existir, o servidor devolverá a seguinte resposta de erro:

```
{"error": "invalid_grant", "error_description": "Bad credentials"}
```

Agora a aplicação `bookserver` começa a caminhar cada vez mais em direção a uma solução profissional. Entretanto, antes de pensar que essa aplicação já pode ser colocada em produção, é preciso avaliar questões de segurança, como a criptografia do token, e questões de performance. Por enquanto, não se preocupe com essas questões, pois elas serão tratadas mais adiante.

13.5 Conclusão

Ao adicionar o suporte ao banco de dados, a aplicação passa a ficar mais próxima de algo que podemos colocar em produção. O interessante é entender como esse tipo de configuração funciona, pois infelizmente a maioria dos exemplos que encontramos em blogs e tutoriais não cobrem problemas reais.

Com o que você já estudou até agora neste livro, tenho certeza de que já possui conhecimento suficiente para usar OAuth 2.0 em suas aplicações com Java. Além do conhecimento do protocolo, à medida que vamos avançando, mais recursos do próprio framework Spring vão sendo explorados, deixando clara a importância de se conhecer as bases usadas para a construção do Spring Security OAuth2.

CAPÍTULO 14

Client com o Spring Security OAuth2

Em todos testes realizados até agora para acessar os recursos protegidos por OAuth 2.0, ou usamos a aplicação `client`, ou fizemos uma requisição direta nos endpoints do OAuth Provider (através do uso do comando `CURL`, ou da ferramenta Postman). A questão é que fomos obrigados a lidar diretamente com o protocolo OAuth 2.0.

Quando mostrei a implementação do Client, tivemos de montar a requisição para o endpoint de consulta de livros da aplicação `bookserver` da seguinte maneira:

```
MultiValueMap<String, String> headers
    = new LinkedMultiValueMap<>();

headers.add("Authorization", "Bearer " + token);

String endpoint = "http://localhost:8080/api/v2/livros";

RequestEntity<Object> request = new RequestEntity<>(
    headers,
    HttpMethod.GET,
    URI.create(endpoint)
);
```

Perceba que foi necessário montar o cabeçalho `HTTP` da requisição de forma manual. Além disso, para obtermos um token de acesso, criamos algumas implementações específicas para cada grant type. Quando usamos o grant type **Authorization Code**, por exemplo, a aplicação precisou interagir com a classe `AuthorizationCodeTokenService`.

Ao implementarmos a classe `AuthorizationCodeTokenService`, foi necessário adicionar todos os parâmetros de formulário e cabeçalhos `HTTP` manualmente, conforme mostrado a seguir:

```

private MultiValueMap<String, String> getBody(String authorizationCode) {
    MultiValueMap<String, String> dadosFormulario
        = new LinkedMultiValueMap<>();

    dadosFormulario.add("grant_type", "authorization_code");
    dadosFormulario.add("code", authorizationCode);
    dadosFormulario.add("scope", "read write");
    dadosFormulario.add("redirect_uri",
        "http://localhost:9000/integracao/callback");

    return dadosFormulario;
}

private HttpHeaders getHeader(BasicAuthentication clientAuthentication) {
    HttpHeaders httpHeaders = new HttpHeaders();

    httpHeaders.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
    httpHeaders.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    httpHeaders.add("Authorization", "Basic " +
clientAuthentication.getCredenciaisBase64());

    return httpHeaders;
}

```

Não seria melhor se o Spring Security OAuth2 nos fornecesse um meio automático de recuperação de token de acesso quando a aplicação tentasse fazer uma requisição para um endpoint protegido por OAuth 2.0? Imagine usar uma implementação de `RestTemplate` da forma mostrada a seguir, e não precisar se preocupar em como obter um token de acesso.

```
Livro[] livros = oAuth2RestTemplate.getForObject(endpoint, Livro[].class);
```

A boa notícia é que o Spring Security OAuth2 já fornece exatamente o que estamos procurando. Você pode se perguntar: mas por que não usamos essa abordagem antes? A resposta é simples: apresentei a abordagem mais difícil para que você entendesse todo o funcionamento do protocolo OAuth 2.0. Dessa forma, você aprendeu em detalhes como interagir com um OAuth Provider.

Neste capítulo, vamos então configurar a aplicação `client` para interagir com a aplicação `bookserver` utilizando o mecanismo de **OAuth 2.0 Client**, oferecido pelo Spring Security OAuth2. Antes de tudo, importe para sua IDE os projetos `bookserver` e `client`, disponíveis no diretório `livro-spring-oauth2/capitulo-14`. Além disso, execute o seguinte comando SQL no banco de dados `clientapp`:

```
UPDATE usuario SET token_bookserver = null;
```

Com o comando SQL que acabei de mostrar, estamos desassociando a conta do usuário na aplicação `client` com a conta existente na aplicação `bookserver`. A seguir, execute o seguinte comando no banco de dados `clientapp` para adicionar mais uma coluna que será usada para manter a data de expiração do token OAuth 2.0. Isso será importante mais adiante quando estivermos usando a implementação de `client` do Spring Security OAuth2.

```
ALTER TABLE usuario ADD COLUMN expiracao_token DATETIME;
```

Os comandos SQL que mostrei aqui também estão disponíveis no arquivo `oauth_client.sql` do projeto `client` que estamos usando. Como adicionamos mais uma coluna na tabela usuário que diz respeito à data de expiração do token de acesso, adicione a propriedade `dataDeExpiracao` na classe `AcessoBookserver` deixando-a conforme vemos a seguir:

```
@Embeddable  
@ToString  
public class AcessoBookserver {  
  
    @Getter @Setter  
    @Column(name = "token_bookserver")  
    private String accessToken;  
  
    @Getter @Setter  
    @Column(name = "expiracao_token")  
    private Calendar dataDeExpiracao;  
  
}
```

Agora, abra o projeto `bookserver` e observe que a classe `ConfiguracaoOAuth2.OAuth2AuthorizationServer` está mantendo as configurações do `client`, ou seja, os *Client Details* em memória. Neste capítulo, usaremos essa abordagem para que seja mais fácil entender como os dados dos Clients estão configurados.

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
    clients.inMemory()
        .withClient("cliente-app")
        .secret("123456")
        .scopes("read", "write")
        .resourceIds(RESOURCE_ID)
        .authorities("read", "write")
        .authorizedGrantTypes(
            "password",
            "authorization_code",
            "refresh_token")
        .accessTokenValiditySeconds(120)

        .and()
        .withClient("cliente-admin")
        .secret("123abc")
        .authorizedGrantTypes("client_credentials")
        .scopes("read")
        .resourceIds(RESOURCE_ID)

        .and()
        .withClient("cliente-browser")
        .secret("abc")
        .authorizedGrantTypes("implicit")
        .scopes("read");
}
```

14.1 Situação atual do Client

Para explorar o uso da configuração do Client, vamos usar o grant type **Authorization Code** por se tratar do fluxo mais completo. A ideia agora é deixar de usar o serviço `AuthorizationCodeTokenService` que criamos no capítulo sobre o grant type Authorization Code, e migrar para a solução disponibilizada pelo Spring Security OAuth2. Antes de iniciar qualquer teste de solicitação de um token, certifique-se de que a coluna `token_bookserver` da tabela `usuario` do banco de dados `clientapp` esteja vazia.

Inicie as aplicações `bookserver` e `client` e, em seguida, acesse a URL da aplicação Client através do link `http://localhost:9000`. Navegue normalmente pela aplicação e autorize o acesso aos livros do usuário logado. Até aqui, tudo deve funcionar perfeitamente. Se você executar uma consulta na tabela `usuario` do banco de dados `clientapp`, deverá perceber que agora a coluna `token_bookserver` deve estar preenchida para o usuário que você usou no teste.

Agora que você já navegou nas aplicações de exemplo, vamos analisar os seguintes pontos relacionados à integração entre a aplicação `client` e a aplicação `bookserver`:

- Quando o usuário acessa a opção `Minha Conta`, os livros que ele possui na aplicação `bookserver` são exibidos caso já exista um token de acesso gerado para a conta do usuário logado.
- Caso o usuário ainda não tenha autorizado a aplicação `client`, ele precisa acessar o link `http://localhost:9000/integracao` para começar o fluxo de autorização do OAuth 2.0.

E se o fluxo de autorização do OAuth 2.0 já pudesse ser iniciado automaticamente caso o token ainda não existisse para o usuário logado? Podemos então utilizar uma classe do tipo `OAuth2RestTemplate` em vez da classe que estamos usando hoje, dentro de `BookserverService`.

A classe `OAuth2RestTemplate` já abstrai todo o processo de autorização e obtenção do token de acesso, eliminando a necessidade de termos de lidar com todas as particularidades do

protocolo OAuth 2.0. Essa extensão de `RestTemplate` parece ser bem interessante, entretanto, para que seja possível o seu uso, precisamos adicionar uma nova dependência e realizar algumas configurações conforme veremos a seguir.

14.2 Iniciando a configuração do Spring Security OAuth2 para o Client

Antes de começar a configurar a aplicação para usar o Spring Security OAuth2, adicione a dependência a seguir no arquivo `pom.xml` do projeto `client`:

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
```

Agora que a dependência foi adicionada, vamos analisar o método `livros` da classe `BookserverService`. E se tentarmos usar a classe `OAuth2RestTemplate` de uma maneira semelhante à que já estamos fazendo atualmente com a classe `RestTemplate`? Substitua o código do método `livros` pelo código mostrado a seguir.

```
public List<Livro> livros(String token) throws
UsuarioSemAutorizacaoException {
    String endpoint = "http://localhost:8080/api/v2/livros";

    OAuth2RestTemplate restTemplate = new OAuth2RestTemplate();

    try {
        Livro[] livros = restTemplate.getForObject(endpoint,
        Livro[].class);
        return listaFromArray(livros);
    } catch (HttpClientErrorException e) {
        throw new UsuarioSemAutorizacaoException("não foi possível obter
os livros do usuário");
    }
}
```

}

Apesar de mais simples, o novo código não compila. O problema é que, para criar uma instância de `OAuth2RestTemplate`, precisamos passar alguns parâmetros para o método construtor. Então, não podemos criar os objetos necessários aqui mesmo e injetar o que for preciso na classe `OAuth2RestTemplate`?

A resposta é não, pois a classe `OAuth2RestTemplate` possui como dependência um objeto do tipo `OAuth2ClientContext`, responsável por manter o contexto de execução dos fluxos OAuth 2.0 por **usuário**. Ou seja, uma instância de `OAuth2ClientContext` é criada por **sessão**.

Para evitar a escrita de código de infraestrutura dentro da classe que contém regra de negócio, não vamos criar uma instância de OAuth2RestTemplate dentro do método livros da classe BookserverService. Em vez disso, usaremos os mecanismos de injeção de dependência do Spring criando uma classe de configuração em que vamos declarar um bean do tipo OAuth2RestTemplate. Crie a classe ConfiguracaoResource dentro do pacote br.com.casadocodigo.integracao.bookserver com o seguinte conteúdo:

```

        Arrays.asList(new
AuthorizationCodeAccessTokenProvider()));

        provider.setClientTokenServices(clientTokenServices);
        template.setAccessTokenProvider(provider);

        return template;
    }

}

```

Com o código que adicionamos dentro da classe `ConfiguracaoResource`, já estamos declarando um `bean` do tipo `OAuth2RestTemplate`. Perceba que agora já estamos providenciando os objetos `resourceDetails` e `oauth2ClientContext` para o método construtor da classe `OAuth2RestTemplate`. Entretanto, ainda precisamos criar o método `bookserver` que retorna um objeto do tipo `OAuth2ProtectedResourceDetails`. Faça-o dentro da classe `ConfiguracaoResource`, conforme mostrado a seguir:

```

@Configuration
public class ConfiguracaoResource {

    // atributos omitidos

    @Bean
    public OAuth2ProtectedResourceDetails bookserver() {
        AuthorizationCodeResourceDetails detailsForBookserver
            = new AuthorizationCodeResourceDetails();

        return detailsForBookserver;
    }

    // resto do código omitido
}

```

Agora o trecho de código que atribui o retorno do método `bookserver` para a referência `resourceDetails` já compila. Mas do que se trata esse `bean` do tipo `OAuth2ProtectedResourceDetails`?

Note que, dentro do método `bookserver`, estamos instanciando um objeto do tipo `AuthorizationCodeResourceDetails`, que implementa `OAuth2ProtectedResourceDetails`. É através dessa classe que definiremos todos os detalhes do recurso protegido que queremos acessar.

Como exemplos de detalhes que serão mantidos na classe `OAuth2ProtectedResourceDetails`, temos as credenciais do Client e os endpoints de autorização e solicitação de tokens. Todos esses dados eram mantidos dentro da classe `AuthorizationCodeTokenService` até agora. Entretanto, ao criar o `bean` mostrado a seguir, todos esses detalhes de configuração estarão disponíveis para uso em qualquer parte do projeto.

```
@Bean
public OAuth2ProtectedResourceDetails bookserver() {
    AuthorizationCodeResourceDetails detailsForBookserver = new
    AuthorizationCodeResourceDetails();

    detailsForBookserver.setId("bookserver");
    detailsForBookserver.setTokenName("oauth_token");
    detailsForBookserver.setClientId("cliente-app");
    detailsForBookserver.setClientSecret("123456");
    detailsForBookserver.setAccessTokenUri(
        "http://localhost:8080/oauth/token");
    detailsForBookserver.setUserAuthorizationUri(
        "http://localhost:8080/oauth/authorize");
    detailsForBookserver.setScope(Arrays.asList("read", "write"));

    detailsForBookserver.setPreEstablishedRedirectUri(
        "http://localhost:9000/integracao/callback");
    detailsForBookserver.setUseCurrentUri(false);

    detailsForBookserver.setClientAuthenticationScheme(
        AuthenticationScheme.header);
    return detailsForBookserver;
}
```

Muitas das propriedades que estamos configurando para o objeto do tipo `AuthorizationCodeResourceDetails` podem ser entendidas intuitivamente. Entretanto, para maiores detalhes, vamos dar uma olhada na definição de cada propriedade através da tabela a seguir.

Propriedade	Descrição
<code>id</code>	Usado internamente pelo Client para identificar o recurso (não é usado no protocolo OAuth 2.0).
<code>tokenName</code>	O nome do token que também é usado apenas internamente pelo Client (não é usado no protocolo OAuth 2.0).
<code>clientId</code>	Identificação do Client no Authorization Server.
<code>clientSecret</code>	Chave secreta recebida no momento do registro do Client. No nosso caso, está definida de forma fixa na configuração dos detalhes do Client no Authorization Server.
<code>accessTokenUri</code>	URI usada para obter o token de acesso (estamos usando o padrão fornecido pelo Spring Security OAuth2).
<code>userAuthorizationUri</code>	URI usada para obter a autorização do usuário para acessar o recurso desejado. Também estamos usando o padrão fornecido pelo Spring Security OAuth2 quando configuramos o OAuth Provider.
<code>scope</code>	Escopos desejados durante a solicitação do token de acesso.

Propriedade	Descrição
<i>preEstablishedRedirectUri</i>	URI de redirecionamento para obtenção do <i>authorization code</i> . Se não for preenchido, o Spring Security OAuth2 utiliza a URI sendo acessada no momento.
<i>useCurrentUri</i>	Permite usar a URI corrente (que estiver sendo acessada quando tentarmos obter um recurso via OAuth 2.0).

Propriedade	Descrição
<i>clientAuthenticationScheme</i>	Qual forma de autenticação do Client no Authorization Server deve ser usada durante a solicitação do token de acesso. Estamos usando <i>AuthenticationScheme.header</i> que vai utilizar <i>HTTP Basic Authentication</i> .

14.3 Usando o banco de dados para manter os dados do token do usuário

A criação do objeto do tipo `OAuth2RestTemplate` ainda não compila devido ao seguinte trecho de código:

```
provider.setClientTokenServices(clientTokenServices);
```

Estamos configurando a propriedade `clientTokenServices` do objeto `provider`, passando como parâmetro uma variável que ainda não foi

definida. Mas o que significa esse provider , afinal?

A variável provider do tipo AccessTokenProviderChain que estamos configurando contém uma cadeia de classes responsáveis por tratar cada fluxo de obtenção de token do OAuth 2.0, ou seja, classes especialistas para cada grant type. Como estamos interessados em usar apenas o grant type **Authorization Code**, esse objeto foi criado da seguinte maneira:

```
AccessTokenProviderChain provider = new AccessTokenProviderChain(  
    Arrays.asList(new AuthorizationCodeAccessTokenProvider()));
```

Entretanto, para que esse provider possa gerenciar o token de acesso gerado para um usuário, ou seja, recuperar e salvar tokens no banco de dados, ele precisa de um objeto que possa abstrair todo o acesso a dados relacionados aos tokens do Client. É através de um objeto do tipo ClientTokenServices que o provider consegue gerenciar todo o processo de uso e de solicitação do token de acesso do usuário logado. Vamos então injetar uma instância de ClientTokenServices na classe ConfiguracaoResource .

```
@Configuration  
public class ConfiguracaoResource {  
    // outros atributos omitidos  
    @Autowired  
    private ClientTokenServices clientTokenServices;  
    // outros métodos omitidos  
}
```

Apesar de estarmos injetando um objeto do tipo ClientTokenServices , ainda não temos uma implementação dessa interface. Para que o gerenciamento dos tokens no banco de dados atual seja feito de forma transparente, precisamos de uma implementação para a interface ClientTokenServices . O Spring Security OAuth2 já fornece uma implementação através da classe JdbcClientTokenServices , que conta com uma estrutura de tabelas específica.

Como já temos a tabela usuario no banco de dados clientapp , vamos criar uma implementação customizada da interface

`ClientTokenServices` que possui os seguintes métodos:

- `getAccessToken` : método que usaremos para buscar o token de acesso existente para o usuário que estiver logado.
- `saveAccessToken` : método que salva os dados do token de acesso obtido durante o fluxo de autorização do OAuth 2.0.
- `removeAccessToken` : método que remove o access token do usuário logado.

Vamos criar a classe `BookserverClientTokenServices` no pacote `br.com.casadocodigo.integracao.bookserver`, conforme mostrado a seguir:

```
@Service
public class BookserverClientTokenServices implements ClientTokenServices {
    @Autowired
    private UsuariosRepository usuarios;
}
```

Em seguida, vamos implementar cada um dos métodos definidos pela interface `ClientTokenServices`. Primeiro adicione o método `getAccessToken`.

```
@Override
public OAuth2AccessToken getAccessToken(
    OAuth2ProtectedResourceDetails resource, Authentication
    authentication) {

    UsuarioLogado usuarioLogado = (UsuarioLogado) authentication
        .getPrincipal();

    Usuario usuario = usuarios.findById(usuarioLogado.getId());

    String accessToken = usuario.getAcessoBookserver().getAcessoToken();
    Calendar dataDeExpiracao = usuario.getAcessoBookserver()
        .getDataDeExpiracao();
```

```

    if (accessToken == null) return null;

    DefaultOAuth2AccessToken oAuth2AccessToken
        = new DefaultOAuth2AccessToken(accessToken);
    oAuth2AccessToken.setExpiration(dataDeExpiracao.getTime());

    return oAuth2AccessToken;
}

```

Basicamente o que o método `getAccessToken` faz é buscar os dados do usuário logado na aplicação e criar uma instância de `DefaultOAuth2AccessToken`, contendo o token e a data de expiração que estiver gravada na tabela `usuario`. O próximo método a ser implementado é o `saveAccessToken`.

```

@Override
public void saveAccessToken(OAuth2ProtectedResourceDetails resource,
                            Authentication authentication, OAuth2AccessToken accessToken) {

    AcessoBookserver acessoBookserver = new AcessoBookserver();
    acessoBookserver.setAcessoToken(accessToken.getValue());

    Calendar expirationDate = Calendar.getInstance();
    expirationDate.setTime(accessToken.getExpiration());

    acessoBookserver.setDataDeExpiracao(expirationDate);

    UsuarioLogado usuarioLogado = (UsuarioLogado)
    authentication.getPrincipal();
    Usuario usuario = usuarios.findById(usuarioLogado.getId());

    usuario.setAcessoBookserver(acessoBookserver);

    usuarios.save(usuario);
}

```

A implementação do método `saveAccessToken` também é bastante direta. Conforme podemos observar, esse método recupera os dados do token que estão contidos no parâmetro `accessToken`, e atribui o novo token e a data de expiração para uma nova instância

de `AcessoBookserver`. Essa nova instância de `AcessoBookserver` então é atribuída ao usuário que estiver logado no sistema. O último método que vamos implementar é o método responsável por remover o token de acesso do usuário no banco de dados.

```
@Override  
public void removeAccessToken(OAuth2ProtectedResourceDetails resource,  
Authentication authentication) {  
    UsuarioLogado usuarioLogado = (UsuarioLogado)  
authentication.getPrincipal();  
    Usuario usuario = usuarios.findById(usuarioLogado.getId());  
  
    usuario.setAcessoBookserver(null);  
    usuarios.save(usuario);  
}
```

Agora já fizemos bastante configurações, mas a aplicação já funciona? Ainda não. Precisamos adicionar a anotação

`@EnableOAuth2Client` na classe `ConfiguracaoResource`. Essa anotação importa a configuração `OAuth2ClientConfiguration`, responsável por definir o tão importante **objeto de contexto** do tipo `OAuth2ClientContext`, que estamos usando em nossa configuração do `OAuth2RestTemplate`.

Abrindo a classe `OAuth2ClientConfiguration` do Spring Security OAuth2, podemos ver como o bean do tipo `OAuth2ClientContext` é criado. Perceba que ele possui uma referência para uma instância da classe `AccessTokenRequest`. A classe `AccessTokenRequest` é necessária para manter os parâmetros usados entre as requisições de autorização e solicitação de token de acesso junto ao Authorization Server.

```
@Configuration  
public class OAuth2ClientConfiguration {  
  
    @Bean  
    public OAuth2ClientContextFilter oauth2ClientContextFilter() {  
        OAuth2ClientContextFilter filter = new  
        OAuth2ClientContextFilter();
```

```

        return filter;
    }

    @Bean
    @Scope(value = "request", proxyMode = ScopedProxyMode.INTERFACES)
    protected AccessTokenRequest accessTokenRequest(@Value("#
{request.parameterMap}")
        // conteúdo omitido
    }

    @Configuration
    protected static class OAuth2ClientContextConfiguration {

        @Resource
        @Qualifier("accessTokenRequest")
        private AccessTokenRequest accessTokenRequest;

        @Bean
        @Scope(value = "session", proxyMode = ScopedProxyMode.INTERFACES)
        public OAuth2ClientContext oauth2ClientContext() {
            return new DefaultOAuth2ClientContext(accessTokenRequest);
        }
    }

}

```

Além disso, essa classe de configuração também define o filtro `OAuth2ClientContextFilter`, que é responsável por tratar todo o mecanismo de redirecionamento do usuário entre o Client e o `AuthorizationServer`. A classe `ConfiguracaoResource` deve ficar com as seguintes anotações:

```

@Configuration
@EnableOAuth2Client
public class ConfiguracaoResource {
    // conteúdo omitido
}

```

Agora sim, todas as configurações estão prontas para injetarmos um bean do tipo `OAuth2RestTemplate` na classe `BookserverService`, conforme mostrado a seguir. Além disso, observe no método `livros` que já não estamos mais tentando criar uma instância de `OAuth2RestTemplate`.

```
@Service
public class BookserverService {

    @Autowired
    private OAuth2RestTemplate restTemplate;

    public List<Livro> livros(String token) throws
    UsuarioSemAutorizacaoException {
        String endpoint = "http://localhost:8080/api/v2/livros";

        try {
            Livro[] livros = restTemplate.getForObject(
                endpoint, Livro[].class);

            return listaFromArray(livros);
        } catch (HttpClientErrorException e) {
            throw new UsuarioSemAutorizacaoException("não foi possível
obter os livros do usuário");
        }
    }

    // resto do código omitido
}
```

Antes de executar a aplicação para testar o fluxo OAuth 2.0, abra a classe `IntegracaoController` e substitua todo seu conteúdo para o código seguinte (remova os métodos e atributos existentes).

```
@Controller
@RequestMapping("/integracao")
public class IntegracaoController {

    @RequestMapping(value = "/callback", method = RequestMethod.GET)
    public ModelAndView callback() {
```

```
        return new ModelAndView("forward:/minhaconta/principal");  
    }  
  
}
```

Agora já não precisamos mais da classe `AuthorizationCodeTokenService`, deixando a `IntegracaoController` bem mais enxuta. Mas por que ainda estamos definindo a classe `IntegracaoController` se ela simplesmente faz um `forward` para `/minhaconta/principal` ?

Esse controlador ainda é necessário, pois estamos usando uma URI de redirecionamento customizada que é `/integracao/callback`. Conforme veremos mais adiante quando estiverem sendo abordadas algumas questões sobre segurança, é importante usar uma URI de redirecionamento específica em vez de se basear apenas no domínio da aplicação `Client`.

Estamos prontos para iniciar os testes do novo `Client`. Certifique-se novamente de que não existe nenhum token salvo na coluna `token_bookserver` da tabela `usuario`, e inicie as aplicações `client` e `bookserver`. Vamos verificar se tudo funcionou como esperávamos.

Para isso, navegue na aplicação e acesse a opção `Minha conta`. Após acessar e passar pelo fluxo de autorização, os livros do usuário devem ser exibidos na tela conforme mostrado a seguir (claramente os dados devem ser diferentes para o usuário que você estiver utilizando).

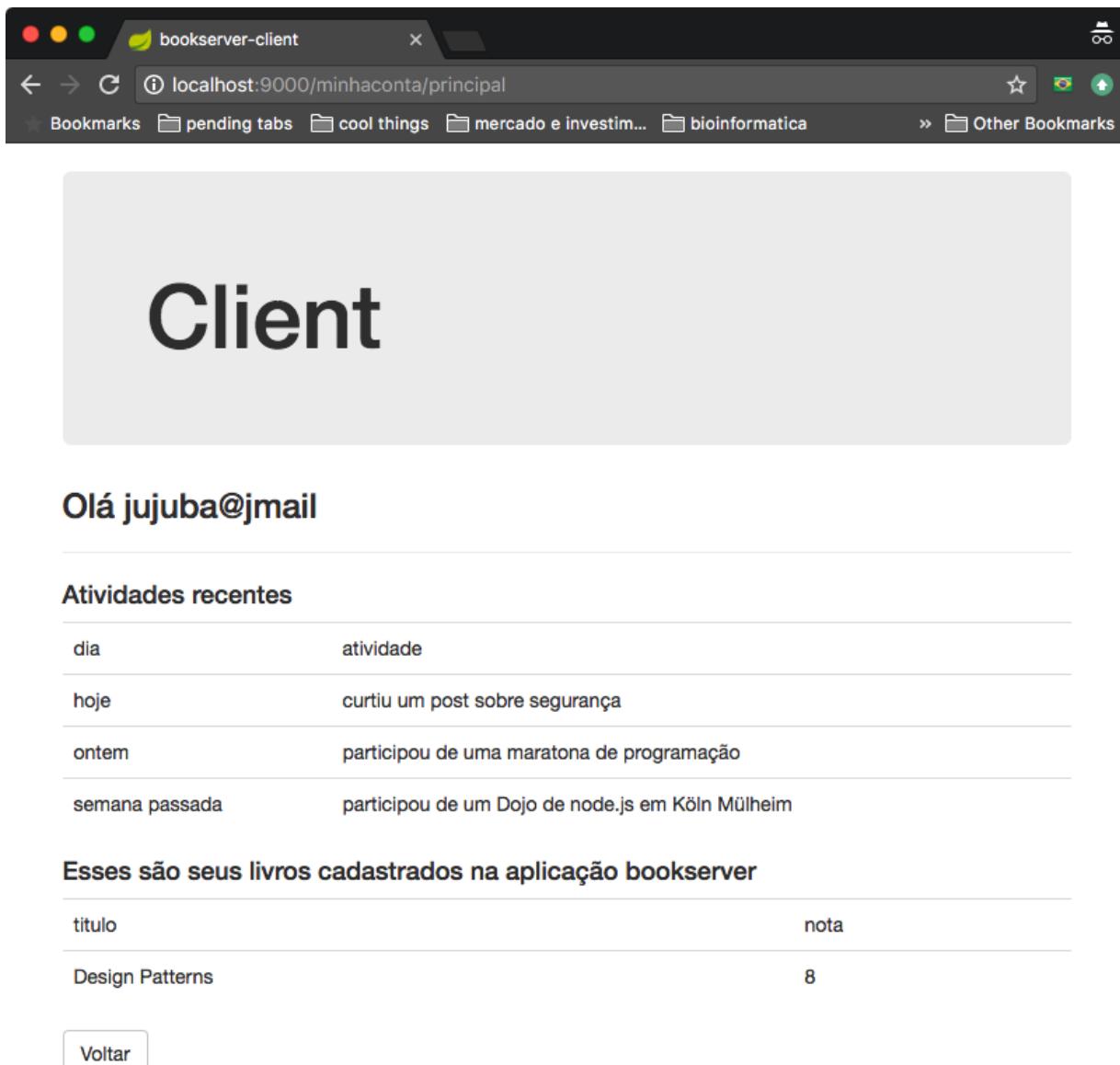


Figura 14.1: Tela que mostra os livros do usuário

Para ver como os dados do token ficaram no banco de dados clientapp , execute o seguinte comando SQL no console do MySQL:

```
SELECT token_bookserver, expiracao_token  
FROM usuario;
```

Ao executar o comando mostrado anteriormente, você deve ver algo semelhante ao resultado a seguir:

token_bookserver	expiracao_token
c171a35e-152c-4a95-abe9-e9059a5c8bf4	2017-04-08 18:34:09
NULL	NULL
NULL	NULL
NULL	NULL

Figura 14.2: Resultado da consulta na tabela de usuários

14.4 Requisição usando OAuth2RestTemplate nos bastidores

Usar a classe `OAuth2RestTemplate` é muito interessante e poupa o desenvolvedor de ter de se preocupar com detalhes do protocolo OAuth 2.0. Ao tentar acessar um recurso protegido por OAuth 2.0 utilizando a classe `OAuth2RestTemplate`, esta abstrai todo o processo de obtenção do token de acesso. A primeira coisa que é realizada internamente é tentar obter o token através da classe que mantém o contexto de uma requisição OAuth 2.0. Esse contexto é gerenciado pela classe `DefaultOAuth2ClientContext` que já estudamos anteriormente.

Após solicitar o token de acesso, o método `doExecute` da classe `OAuth2RestTemplate` prossegue sua execução criando um objeto que representa a requisição do tipo `AccessTokenRequest`. A questão é que, para realizar a requisição para o endpoint protegido por OAuth 2.0, precisamos de um token de acesso válido. Portanto, antes de realizar a requisição para o endpoint, é realizada uma verificação quanto à validade do token e se ele realmente existe no banco de dados.

Caso o token não existir ou for inválido, a própria classe `OAuth2RestTemplate` inicia o processo de solicitação do token de

acesso através do grant type configurado. Mas como essa implementação de `restTemplate` sabe qual o grant type deve ser usado? Quando criamos a definição do bean `OAuth2RestTemplate`, já informamos quais grant types são suportados, lembra?

```
AccessTokenProviderChain provider = new
AccessTokenProviderChain(Arrays.asList(
    new AuthorizationCodeAccessTokenProvider()));
```

Internamente, o que a classe `OAuth2RestTemplate` faz é solicitar o token para o objeto `provider` que foi referenciado durante sua criação.

```
accessToken = accessTokenProvider.obtainAccessToken(resource,
accessTokenRequest);
```

No fim das contas, o objeto `provider` usado pela classe `OAuth2RestTemplate` delegará a obtenção do token para o objeto especializado no grant type sendo usado. No nosso caso, estamos usando `AuthorizationCodeAccessTokenProvider` que, por sinal, é a classe que vai usar as configurações que definimos pela classe `AuthorizationCodeResourceDetails`.

Durante o fluxo de obtenção do token, caso este não seja válido, o provider especializado no grant type em questão lançará uma exceção do tipo `UserRedirectRequiredException` que contém todos os dados necessários para realização do redirecionamento. É interessante ver como tudo que configuramos interage entre si durante uma requisição. Isso nos ajuda a entender melhor como realizar as configurações corretamente.

14.5 Conclusão

Este capítulo mostrou como podemos interagir com um endpoint protegido por OAuth 2.0 utilizando os mecanismos de Client, oferecidos pelo Spring Security OAuth2. Agora você sabe como

configurar um Client e como usar o banco de dados para manter os dados do token de acesso dos usuários dele.

Do jeito que configuramos a aplicação, caso o token de acesso expire, o usuário terá de passar por todo o fluxo de obtenção de token definido no grant type Authorization Code. Caso queira usar um `refresh_token` para melhorar a experiência do usuário, tudo o que você precisa fazer é adicionar um atributo para o `refresh_token` na classe `AcessoBookserver` e adicionar uma nova coluna na tabela `usuario`.

Além disso, você também precisa fazer com que a classe `BookserverClientTokenServices` passe a ler e salvar esse novo atributo. Com os conhecimentos que você já adquiriu até agora, realize esse exercício antes de partir para o próximo capítulo.

No próximo capítulo, vamos estudar um assunto interessante que nos ajudará a otimizar a forma como o Resource Server faz para validar um token de acesso.

CAPÍTULO 15

Alternativas para validação de tokens

Pensando na aplicação `bookserver` em seu estado atual, a arquitetura do OAuth Provider conta com o Resource Server e o Authorization Server configurados em uma única aplicação. Existe um acoplamento bem forte entre esses dois componentes do OAuth 2.0, que se mostra maior quando consideramos o meio de validação de tokens de acesso sendo usado por parte do Resource Server.

Conforme vimos no capítulo *Funcionamento interno do Spring Security OAuth2*, quando um recurso protegido por OAuth 2.0 é acessado, um filtro que foi configurado pelo Resource Server intercepta a chamada, delegando a validação do token para a classe `DefaultTokenServices`. Esta, por sua vez, utiliza um `TokenStore` para consultar detalhes sobre o token recebido na requisição.

No caso da aplicação `bookserver`, até agora usamos dois tipos de `TokenStore`: `InMemoryTokenStore` e `JdbcTokenStore`. Usar `JdbcTokenStore` pode gerar um grande número de acessos ao banco de dados caso a API protegida por OAuth 2.0 seja muito utilizada. Quanto ao `InMemoryTokenStore`, ele não é um modelo indicado para um sistema que será colocado em produção, pois não queremos que os tokens gerados sejam perdidos toda vez que o servidor precisar ser reiniciado.

Já que conhecemos o processo de acesso aos dados do token, como a validação acontece efetivamente? Quais dados precisamos conhecer a respeito do token para verificar se um é válido? O processo de validação considera dois aspectos principais a respeito de um token de acesso, conforme podemos ver no trecho de código

do método `loadAuthentication` da classe `DefaultTokenServices` (classe interna do Spring Security OAuth2).

```
public OAuth2Authentication loadAuthentication(String accessTokenValue)
    throws AuthenticationException, InvalidTokenException {
    OAuth2AccessToken accessToken = tokenStore
        .readAccessToken(accessTokenValue);

    if (accessToken == null) {
        throw new InvalidTokenException("Invalid access token: " +
accessTokenValue);
    }
    else if (accessToken.isExpired()) {
        tokenStore.removeAccessToken(accessToken);
        throw new InvalidTokenException("Access token expired: " +
accessTokenValue);
    }
    // resto do código omitido
}
```

Esse trecho de código mostra que o processo de validação verifica se o token recebido existe (não importando onde este é armazenado) e se expirou (se a data de expiração dele é maior que a data atual). Veja que precisamos conhecer o atributo `expires_in` do token para verificar se ele ainda está ativo.

Considerando uma aplicação simples e que não recebe muitos acessos em sua API protegida por OAuth 2.0, a arquitetura que estamos usando pode funcionar perfeitamente. Entretanto, em uma arquitetura mais robusta em que os componentes do OAuth Provider são distribuídos em máquinas ou instâncias separadas, nem sempre o Resource Server poderá acessar o banco de dados do Authorization Server diretamente. Aliás, a estrutura separada do OAuth Provider é bastante indicada até mesmo por questões de segurança, pois reduz a superfície de ataque ao servidor OAuth.

Nesse caso, como o Resource Server pode validar um token de acesso sem acessar diretamente o banco de dados onde os tokens são armazenados? Neste capítulo, vamos resolver justamente esse

tipo de problema. Estudaremos duas estratégias diferentes para validação de tokens de acesso, sendo uma através de uma API para recuperação dos dados associados ao token (introspecção de tokens) e outra forma através de tokens autocontidos (JWT - *JSON Web Token*).

Vamos utilizar o código-fonte da aplicação `bookserver` que está disponível no diretório `livro-spring-oauth2/capitulo-15`. Portanto, já importe esse projeto para sua IDE, pois neste capítulo teremos mais prática.

15.1 Introspecção de tokens

Para recuperação dos dados associados a um token de acesso (metadados), já existe uma especificação que define uma maneira para que o Resource Server possa consultar o estado de um token, bem como recuperar os metadados dele. Tal especificação define esse mecanismo como **introspecção de tokens**, que pode ser consultada para maiores detalhes em <https://tools.ietf.org/html/rfc7662>.

O mecanismo de introspecção de tokens basicamente conta com um endpoint em que o Resource Server informa um token como parâmetro de requisição, e recebe um JSON como resposta contendo os metadados (entre eles, um atributo obrigatório chamado `active`). Vale lembrar de que a comunicação com esse endpoint também deve ser realizada através de TLS/SSL e o Resource Server precisa ser autenticado.

Um ponto interessante a respeito do endpoint de introspecção de tokens é que ele pode ser usado para validar qualquer tipo de token OAuth 2.0, ou seja, até mesmo *refresh tokens*. Porém, como deve ser a requisição para esse endpoint?

15.2 Introspecção com o Spring Security OAuth2

Em primeiro lugar, a definição de qual será o endpoint pode ficar a cargo do desenvolvedor decidir. Entretanto, o Spring Security OAuth2 já fornece um meio para introspecção através do endpoint `/oauth/check_token`.

O endpoint para verificação do token não fica disponível por padrão. Para que este possa ser utilizado, precisamos adicionar a seguinte configuração na classe `ConfiguracaoOAuth2.OAuth2AuthorizationServer`.

```
@EnableAuthorizationServer
protected static class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {
    // códigos omitidos
    @Override
    public void configure(AuthorizationServerSecurityConfigurer security)
throws Exception {
    security.checkTokenAccess("hasAuthority('introspection')");
}
    // códigos omitidos
}
```

Por meio da configuração mostrada anteriormente, estamos informando que, para acessar o endpoint `/oauth/check_token`, é necessário que a conta sendo usada tenha a `role introspection`. No código mostrado, informamos a role customizada `introspection`, pois por padrão a classe `AuthorizationServerSecurityConfigurer` possui a propriedade `checkTokenAccess` configurada como `denyAll()`, conforme vemos no trecho de código a seguir:

```
public final class AuthorizationServerSecurityConfigurer extends
    SecurityConfigurerAdapter<DefaultSecurityFilterChain,
HttpSecurity> {
    // outros atributos omitidos
    private String checkTokenAccess = "denyAll()";
    // código omitido
}
```

Embora sabemos qual a role necessária, pois nós mesmos a definimos como `introspection`, como identificamos qual é o usuário que está acessando o endpoint `/oauth/check_token`? Se você respondeu que é pela **autenticação**, acertou. Mas como estamos pensando em um cenário no qual o Resource Server vai validar o token, como configuraremos as credenciais do Resource Server?

Para configurar as credenciais do Resource Server, o Spring Security OAuth2 fornece o mesmo meio que utilizamos para configurar os detalhes do Client. Isso pode parecer um pouco estranho, já que estamos misturando dados de Client com dados de Resource Server, entretanto, o que diferenciará semanticamente um papel do outro será a *role* que vamos configurar para o Resource Server.

A título de curiosidade, para acessar o endpoint `/oauth/check_token` no Spring Security OAuth2, é necessário se autenticar utilizando *HTTP Basic Authentication* e informando `client_id` e `client_secret`. Internamente, o framework trata a autenticação como se realmente estivesse se tratando de um Client. Veja a seguir como a classe `RemoteTokenServices` cria o cabeçalho de autenticação `HTTP` para ser usado durante o acesso ao endpoint `/oauth/check_token`.

```
public class RemoteTokenServices
    implements ResourceServerTokenServices {

    private String getAuthorizationHeader(
        String clientId, String clientSecret) {

        if(clientId == null || clientSecret == null) {
            logger.warn("Null Client ID or Client Secret detected.
Endpoint that requires authentication will reject request with 401
error.");
        }

        String creds = String.format("%s:%s", clientId, clientSecret);
        try {
            return "Basic " + new
String(Base64.encode(creds.getBytes("UTF-8")));
        }
    }
}
```

```

        }
        catch (UnsupportedEncodingException e) {
            throw new IllegalStateException("Could not convert String");
        }
    }
}

```

Vamos então adicionar uma entrada de detalhes de Client com semântica de Resource Server. Para isso, editaremos novamente a classe `ConfiguracaoOAuth2.0Auth2AuthorizationServer`. Altere o método no qual configuramos os *Client Details*, conforme mostrado a seguir:

```

@EnableAuthorizationServer
protected static class OAuth2AuthorizationServer
    extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("cliente-app")
            ...
            // outras configurações para os clients omitidas

            .and()
            .withClient("resource-server")
            .secret("abcd")
            .authorities("introspection");
    }

}

```

Na configuração mostrada no código anterior, não removemos as configurações já existentes. Apenas adicionamos uma configuração no final, onde o "client" possui as credenciais `resource-server` com a senha `abcd` e role de `introspection`.

15.3 Testando o endpoint check_token

Agora já temos condições de testar como funciona o endpoint `/oauth/check_token`. Para isso, inicie a aplicação `bookserver` e crie um token de acesso através do grant type `password`, conforme mostrado a seguir (não esqueça de usar um `username` que exista na aplicação `bookserver`).

```
curl -X POST -H "Authorization: Basic Y2xpZW50ZS1hcHA6MTIzNDU2" \
-H "Content-Type: application/x-www-form-urlencoded" \
-H "Accept: application/json" \
-d
'grant_type=password&username=oauth@mailinator.com&password=123&scope=read
write' \
"http://localhost:8080/oauth/token"
```

Após executar esse comando, você deve receber uma resposta semelhante à mostrada a seguir:

```
{
  "access_token": "0e0683f8-6172-4877-884c-8436d378a78c",
  "token_type": "bearer",
  "refresh_token": "376bdbe2-aeab-4c07-b457-dcdc4a09b98a",
  "expires_in": 94,
  "scope": "read write"
}
```

Agora vamos validar o `access_token` recebido pelo endpoint `/oauth/check_token` se autenticando como `resource-server`, conforme mostrado no comando `CURL`. Substitua o valor do parâmetro `token` pelo obtido na saída da execução do comando anterior.

```
curl -X GET --user resource-server:abcd \
"http://localhost:8080/oauth/check_token?token=0e0683f8-6172-4877-884c-
8436d378a78c"
```

A resposta para esse comando deve ser semelhante à mostrada a seguir:

```
{  
    "aud": [  
        "books"  
    ],  
    "exp": 1491750172,  
    "user_name": "oauth@mailinator.com",  
    "authorities": [  
        "ROLE_USUARIO_COMUM",  
        "read"  
    ],  
    "client_id": "cliente-app",  
    "scope": [  
        "read",  
        "write"  
    ]  
}
```

Repare que temos informações suficientes para descobrir se o token ainda não expirou e se ele existe, além de trazer detalhes sobre o usuário que autorizou a geração do token de acesso, que nesse caso foi o usuário cujo e-mail é `oauth@mailinator.com`. Se tentarmos acessar esse mesmo endpoint passando um token que não existe, teremos uma saída diferente.

```
{  
    "error": "invalid_token",  
    "error_description": "Token was not recognised"  
}
```

Apesar de estarmos realizando um processo de introspecção do token, o Spring Security OAuth2 de fato implementa a especificação de introspecção? Comparando a forma como podemos fazer introspecção com o Spring Security OAuth2 e a forma definida pela especificação, já podemos notar algumas diferenças:

- Em primeiro lugar, a especificação sugere que seja realizada uma requisição HTTP `POST` para o endpoint de introspecção (e estamos usando `GET`).

- A estrutura de dados retornada deve conter o atributo booleano `active` segundo a especificação. A estrutura que recebemos quando acessamos o endpoint `/oauth/check_token` não devolve tal atributo, mas ainda assim nos permite inferir se um token está ou não ativo.
- Outra questão, por sinal bem importante, se deve ao retorno recebido quando o token não existe ou simplesmente não é válido por qualquer outro motivo. A especificação sugere que, quando o token estiver inválido ou não existir, o servidor deve retornar uma resposta contendo apenas o atributo `active` com o valor `false`. Além disso, o status HTTP a ser usado deve ser o 200 em vez do 400, como acontece no Spring Security OAuth2.

Existem propostas no projeto do Spring Security OAuth2 para adequar o esquema de introspecção de tokens à especificação atualizada, conforme o *Pull Request* que pode ser acessado em <https://github.com/spring-projects/spring-security-oauth/pull/464>. Entretanto, até o momento da escrita deste livro, ainda não tínhamos um suporte à introspecção que seguisse a especificação descrita através da RFC 7662.

15.4 Como usar a validação remota

Agora que já temos a configuração de introspecção de tokens, já podemos fazer com que o Resource Server realize a validação do token de forma remota. Para isso, precisamos usar uma classe que já foi citada anteriormente, chamada `RemoteTokenServices`. E como faremos para que o Resource Server utilize-a para validar o token de acesso através do endpoint `oauth/check_token`?

Precisamos definir um `bean` do tipo `RemoteTokenServices` dentro da classe de configuração do Resource Server, que no nosso caso é implementada pela classe `ConfiguracaoOAuth2.OAuth2ResourceServer`.

Adicione então o método `remoteTokenServices` na classe interna `OAuth2ResourceServer`.

```
@EnableResourceServer
protected static class OAuth2ResourceServer
    extends ResourceServerConfigurerAdapter {

    @Bean
    public RemoteTokenServices remoteTokenServices() {
        RemoteTokenServices remoteTokenServices = new
        RemoteTokenServices();
        remoteTokenServices.setClientId("resource-server");
        remoteTokenServices.setClientSecret("abcd");

        remoteTokenServices.setCheckTokenEndpointUrl("http://localhost:8080/oauth/
        check_token");

        remoteTokenServices.setAccessTokenConverter(accessTokenConverter());
        return remoteTokenServices;
    }
}
```

Estamos criando a definição do bean `RemoteTokenServices` informando que devem ser usadas as credenciais que criamos para o Resource Server. Além disso, também precisamos informar qual endpoint será usado para recuperar os metadados do token (ou melhor dizendo, o endpoint para fazer a introspecção do token). Existe mais uma configuração que está sendo adicionada que conta com o método `accessTokenConverter` que, por sinal, ainda não existe.

Precisamos informar para o `remoteTokenServices` como será feita a conversão dos dados do token para o modelo que estamos usando, para identificar um usuário logado no sistema. Se não definirmos tal conversor, o objeto `authentication` que obtemos por meio do contexto do Spring Security vai trazer como `Principal` apenas o e-mail do usuário que autorizou a geração do token que estamos validando.

O que queremos na verdade é que o `Principal` seja um objeto que representa um usuário do sistema (que no nosso caso é definido através da classe `ResourceOwner`). Vamos dar o suporte ao tipo de conversão que desejamos adicionando os seguintes métodos na classe interna `OAuth2ResourceServer`.

```
@Bean
public AccessTokenConverter accessTokenConverter() {
    DefaultAccessTokenConverter converter = new
DefaultAccessTokenConverter();
    converter.setUserTokenConverter(userAuthenticationConverter());
    return converter;
}

@Bean
public UserAuthenticationConverter userAuthenticationConverter() {
    DefaultUserAuthenticationConverter converter
        = new DefaultUserAuthenticationConverter();
    converter.setUserDetailsService(userDetailsService);
    return converter;
}
```

Perceba também que, para criar a classe `UserAuthenticationConverter`, precisamos configurar um `UserDetailsService` para o `converter`. Nós já temos uma implementação de `UserDetailsService` que, por sinal, se trata da classe `DadosDoUsuarioService`. Adicione então o atributo a seguir na classe `OAuth2ResourceServer`.

```
@Autowired
private DadosDoUsuarioService userDetailsService;
```

Agora, ao iniciar a aplicação e acessar o endpoint de consulta de livros, o Resource Server realizará a validação do token de forma remota através do endpoint de introspecção de tokens.

15.5 Quando usar a abordagem de validação remota

Agora, ao gerar um token de acesso e tentar acessar o endpoint de consulta de livros da aplicação `bookserver`, a validação do token será feita pelo objeto `RemoteTokenServices`, poupando o acesso direto ao banco de dados. Com o nível de desacoplamento que atingimos, é possível separar o Resource Server e o Authorization Server em aplicações distintas. Entretanto, apesar de não onerarmos o banco de dados quando a aplicação tiver muitos acessos no endpoint de consulta de livros, passamos a onerar outro recurso importante: a rede.

A própria documentação do Spring Security OAuth2 avisa que o modelo que acabamos de estudar é conveniente caso **não exista** um enorme volume de tráfego nos Resource Servers, pois nesse tipo de situação podemos acabar deixando a aplicação indisponível, violando com isso o princípio de **disponibilidade** que estudamos no primeiro capítulo no tópico *Como as APIs são protegidas*.

Uma forma de melhorar o uso da abordagem remota é utilizar cache no endpoint `/oauth/check_token`. O importante é que, nesse caso, o tempo de validade do cache seja curto o suficiente em relação ao tempo de validade do token. A configuração do tempo de expiração do cache citada deve ajudar a evitar que seja mantido um token que já não é mais válido no cache.

15.6 Tokens autocontidos com JWT

Ambas abordagens que conhecemos até agora para validação de tokens possuem desvantagens que podem onerar o banco de dados ou a rede. Existe algum tipo de alternativa na qual o Resource Server não precise perguntar nada para o Authorization Server a respeito da validade de um token? E se o próprio token pudesse ser gerado a partir dos dados que precisamos, como o e-mail do Resource Owner e o período de validade do token?

É exatamente isso que a especificação JWT (<https://jwt.io/introduction>) oferece como solução. Através de um JWT, podemos transportar dados em formato JSON de maneira compactada através de uma `string` codificada em Base 64. Quais dados são suficientes para identificar um token de acesso? O que são esses dados? JWT define cada um dos atributos que identificam um token como *claims*.

IDENTIDADE BASEADA EM CLAIMS

Uma *claim* nada mais é do que uma expressão que define o que um determinado objeto é. Tenha como objeto qualquer entidade que possa ser identificada como uma pessoa ou um sistema. Para facilitar o entendimento sobre o significado de *claim*, traduzindo essa palavra para o português, podemos interpretar como uma reivindicação ou afirmação que um objeto faz sobre si mesmo ou sobre outro objeto.

Para identificar um token, basicamente precisamos de dados como o tempo de expiração do token, a audiência (em qual Resource Server deve ser usado), quem autorizou a geração do token ou até mesmo quem gerou o token (no caso um identificador do Authorization Server que gerou o token). A especificação JWT classifica as *claims* em três:

- `registered` : são as *claims* definidas na própria especificação conforme veremos logo adiante. Os atributos **registrados** que identificam um token JWT, segundo a especificação, são os seguintes: `iss` , `sub` , `aud` , `exp` , `nbf` , `iat` e `jti` (não se preocupe com esses nomes agora, pois veremos a definição de cada um logo a seguir). As *claims* registradas não são obrigatórias, porém, pelo fato de serem padronizadas, ajudam a manter a interoperabilidade entre sistemas que usam JWT.
- `public` : são atributos que podem ser definidos por uma aplicação e que podem ser usados fora do ambiente dessa

aplicação. Como se tratam de atributos públicos, é interessante que estes sejam registrados para evitar colisão de nomes. O processo para registrar um novo *claim* pode ser verificado na seção 10.1 da RFC 7519, disponível através do link <https://tools.ietf.org/html/rfc7519#section-10.1>.

- **private** : *claims* privados são nomes definidos por uma empresa ou um Authorization Server e que são conhecidos pelas aplicações que os utilizam. Nesse caso, um Resource Server já deve ter conhecimento dos *claims* privados definidos por um Authorization Server. Como exemplo de *claim* privado, podemos usar um atributo que contenha o e-mail do usuário que autorizou a geração do token.

Para que você possa entender o propósito de cada *claim* registrado pela especificação JWT, veja a tabela com o nome e o significado de cada propriedade.

Sigla	Claim	Descrição
<i>iss</i>	Issuer	Indica o criador do token JWT. No nosso caso, foi o Authorization Server da aplicação <i>bookserver</i> .
<i>sub</i>	Subject	Identifica para quem o token foi gerado. Geralmente é associado ao Resource Owner que deu a autorização para geração do token. No caso da aplicação <i>bookserver</i> , podemos usar o e-mail do Resource Owner.
<i>aud</i>	Audience	Indica quem pode aceitar esse token. Geralmente indica para qual Resource Server o token foi gerado.
<i>exp</i>	Expiration time	Tempo de validade do token em segundos.

Sigla	Claim	Descrição
<i>nbf</i>	Not before	Indica um timestamp de quando o token passará a ser válido.

Sigla	Claim	Descrição
<i>iat</i>	Issued at	Timestamp que informa quando o token foi gerado.
<i>jti</i>	JWT ID	Um identificador único para o token.

Estrutura de um JWT

Agora você já tem uma ideia do que pode estar contido em um token JWT, mas todos esses conceitos ainda podem estar vagos. Como é que podemos representar todos os *claims* de que falamos anteriormente através de um token? O Client agora precisa passar um JSON em vez de uma sequência de caracteres conforme viemos usando até agora? Nada disso.

Um token OAuth 2.0 será sempre opaco para o Client, ou seja, o Client nunca precisará saber nada sobre a estrutura do token ou como ele foi gerado. O Client continuará utilizando um token representado através de uma cadeia de caracteres compacta que pode ser transmitida como parâmetro de URL. Vamos entender como todos esses dados em formato JSON são estruturados em um token representado através de uma simples string .



Figura 15.1: Estrutura de um token JWT

A figura anterior mostra a estrutura básica de um token JWT. Conforme podemos ver, um token JWT é formado por três seções separadas por ponto (.), constituídas por um *header*, um *payload* e um *signature*. O *header*, ou cabeçalho, contém informações a respeito do tipo do token e como ele foi assinado ou criptografado.

Mas qual a diferença entre assinar e criptografar um token? Quando assinamos um conteúdo, geramos um hash que serve para garantir a integridade do conteúdo recebido. A especificação JWS ([JSON Web Signature](#)) define um caso especial de JWT que é usada para definir toda a estrutura de um JWT assinado.

Em contrapartida, quando criptografamos um conteúdo, estamos protegendo os dados de serem lidos por usuários ou sistemas não autorizados. Para o caso de JWT criptografado, devemos contar com a especificação JWE ([JSON Web Encryption](#)) que também se trata de uma especialização do JWT. Veremos mais detalhes sobre assinatura e criptografia mais adiante.

A questão é que, para definir se um token foi assinado ou criptografado e como isso foi realizado, a especificação JWT conta com outras especificações complementares (JWS, JWE, JWA e JWK) compondo uma coleção conhecida como JOSE (*JavaScript Object Signing & Encryption*). Portanto, é comum encontrar definições do cabeçalho do JWT como **JOSE header**.

Vejamos um exemplo de cabeçalho em que estamos definindo um JWT que não é assinado nem criptografado.

```
{  
  "typ": "JWT",  
  "alg": "none"  
}
```

Como estamos tratando de um JWT não assinado, o atributo `alg` do cabeçalho foi definido com o valor `none`.

A segunda seção do token contém o *payload* do JWT. É no *payload* que estão contidos os dados sobre o usuário e sobre o token em si, ou seja, são as *claims* de que falamos anteriormente. Essa estrutura, assim como o cabeçalho, também é representada através de um JSON, conforme vemos a seguir:

```
{  
    "iss": "bookserver",  
    "sub": "jujuba@mailinator.com",  
    "name": "Princesa Jujuba",  
    "exp": 1491766663  
}
```

A última seção contém a assinatura do *payload*, ou seja, uma string que pode ser usada para verificar a integridade do *payload*. Através de uma assinatura, podemos garantir que uma mensagem não foi alterada no meio do caminho. Isso poderia ser desastroso para um Resource Server, pois ele poderia acabar aceitando um token gerado por qualquer usuário mal-intencionado em vez do Authorization Server.

Já temos todos os dados necessários para compor cada seção de um token JWT, entretanto, para que ele possa ser transmitido através de uma URL, é necessário codificar cada seção em formato JSON com Base 64. Veja a seguir como poderíamos gerar um exemplo de token usando Java puro. Caso queira testar esse trecho de código, importe o projeto `testes-jwt` que se encontra no diretório `livro-spring-oauth2/capitulo-15`.

```
public class TesteJwt {  
    public static void main(String[] args) {  
        String header = new String(  
            Base64.getEncoder().encode(getHeaderInBytes()));  
  
        String payload = new String(  
            Base64.getEncoder().encode(getPayloadInBytes()));  
  
        String token = header + "." + payload + ".";
```

```

        System.out.println(token);
    }

    private static byte[] getHeaderInBytes() {
        return getFileContentInBytes("header.json");
    }

    private static byte[] getPayloadInBytes() {
        return getFileContentInBytes("payload.json");
    }

    // código omitido
}

```

Considerando o header e o payload que usamos anteriormente, ao executar o código de teste para geração de token, obteremos o seguinte JWT:

ewogICJ0eXAiOiAiSldUIiwKICAiYWxnIjogIm5vbmlUiCn0K.ewogICJpc3Mi0iAiYm9va3Nlc
nZlciIsCiAgInN1YiI6ICJqdWp1YmFAbWFpbGluYXRvcj5jb20iLAogICJuYW1IiogIlByaW5
jZXNhIEp1anViYSIsCiAgImV4cCI6IDE0OTE3NjY2NjMKfQo=.

Perceba que, para gerar o token, basicamente serializamos o conteúdo usando codificação Base 64 nos permitindo chegar a um conteúdo compacto. No exemplo citado, nosso token se trata de um JWT compactado como JWS.

Assinatura e validação do conteúdo

Observando o token gerado anteriormente, podemos notar que a última seção do JWT não foi gerada, ou seja, não assinamos o payload do token. Qual o problema de não assinar o token?

Como a validação do token não conta mais com uma verificação no banco de dados do Authorization Server, qualquer usuário poderia alterar o payload, e o Resource Server poderia aceitar um token contendo *claims* diferentes das que foram definidas pelo Authorization Server. Dessa forma, seria muito fácil usar um token de outro usuário para realizar operações em seu nome.

A assinatura resolve justamente o problema de integridade do payload. Mas como podemos assinar o payload? E o que significa assinar um conteúdo?

Conforme já falei anteriormente, um token assinado é um tipo específico de JWT conhecido como JWS. Fazendo uma analogia com programação orientada a objetos, o JWT seria uma classe abstrata enquanto que um JWS seria a classe concreta que estende JWT. A especificação do JWS (RFC 7515, que pode ser vista em <https://tools.ietf.org/html/rfc7515>) define como um token pode ser assinado, bem como a sua forma de serialização.

Analisando a imagem a seguir, podemos notar que uma assinatura é gerada a partir de um algoritmo, uma chave e o próprio conteúdo (*payload*) a ser assinado. A partir de duas entradas, o algoritmo gera uma assinatura que também pode ser chamada de *tag* em algumas literaturas. A tag gerada é usada para compor a terceira seção do token JWS.

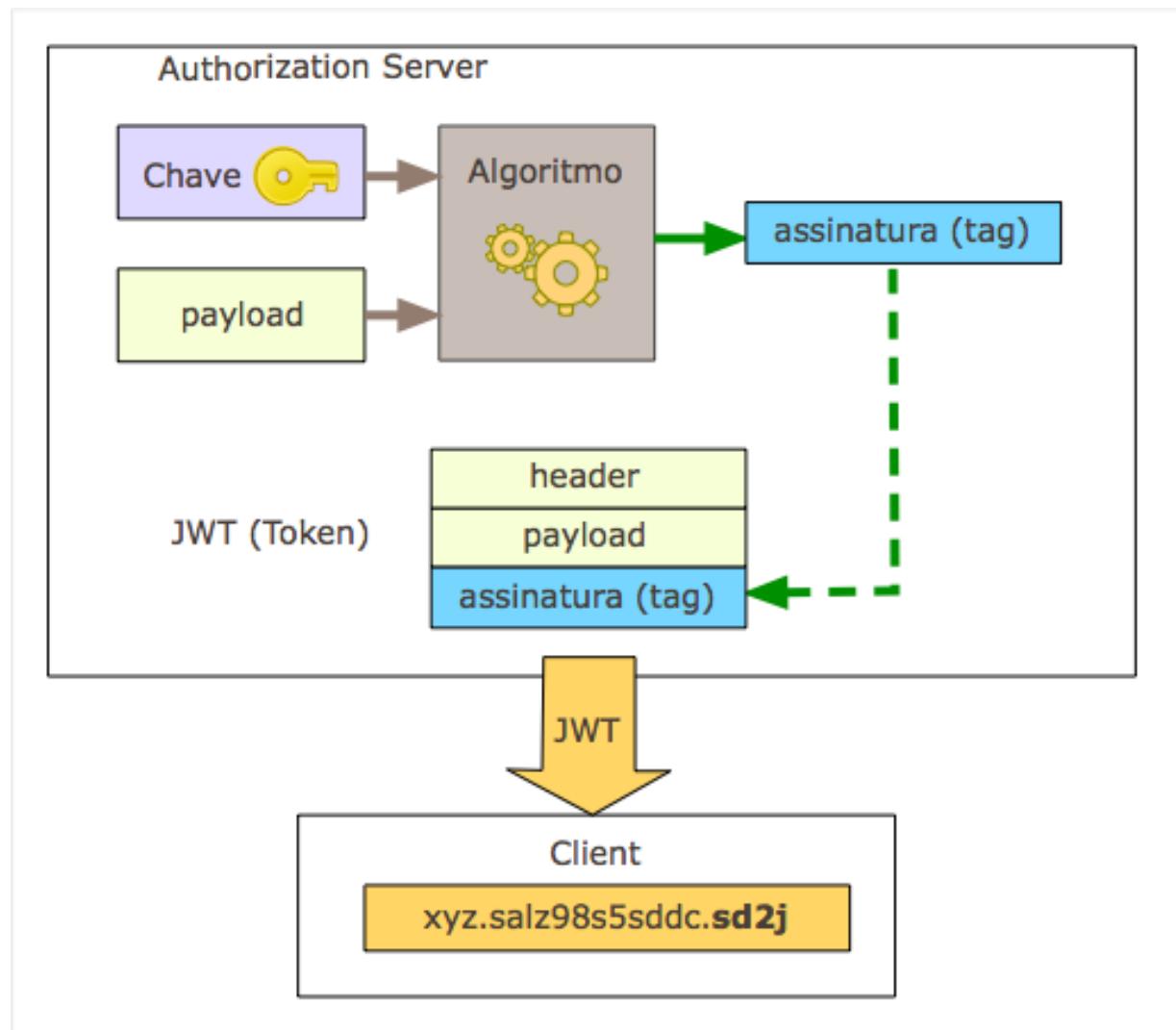


Figura 15.2: Assinatura de um payload

Conforme podemos notar, a assinatura deve ser gerada no Authorization Server que é o componente OAuth que possui tanto os dados para compor o payload quanto a chave de criptografia para assinatura. Em contrapartida, a chave para assinar um conteúdo não deve estar disponível para o Client. Perceba também que o token entregue ao Client continua sendo uma `string`, que pode ser enviada através do cabeçalho `http Authorization` da mesma forma que fazemos com outros tipos de token.

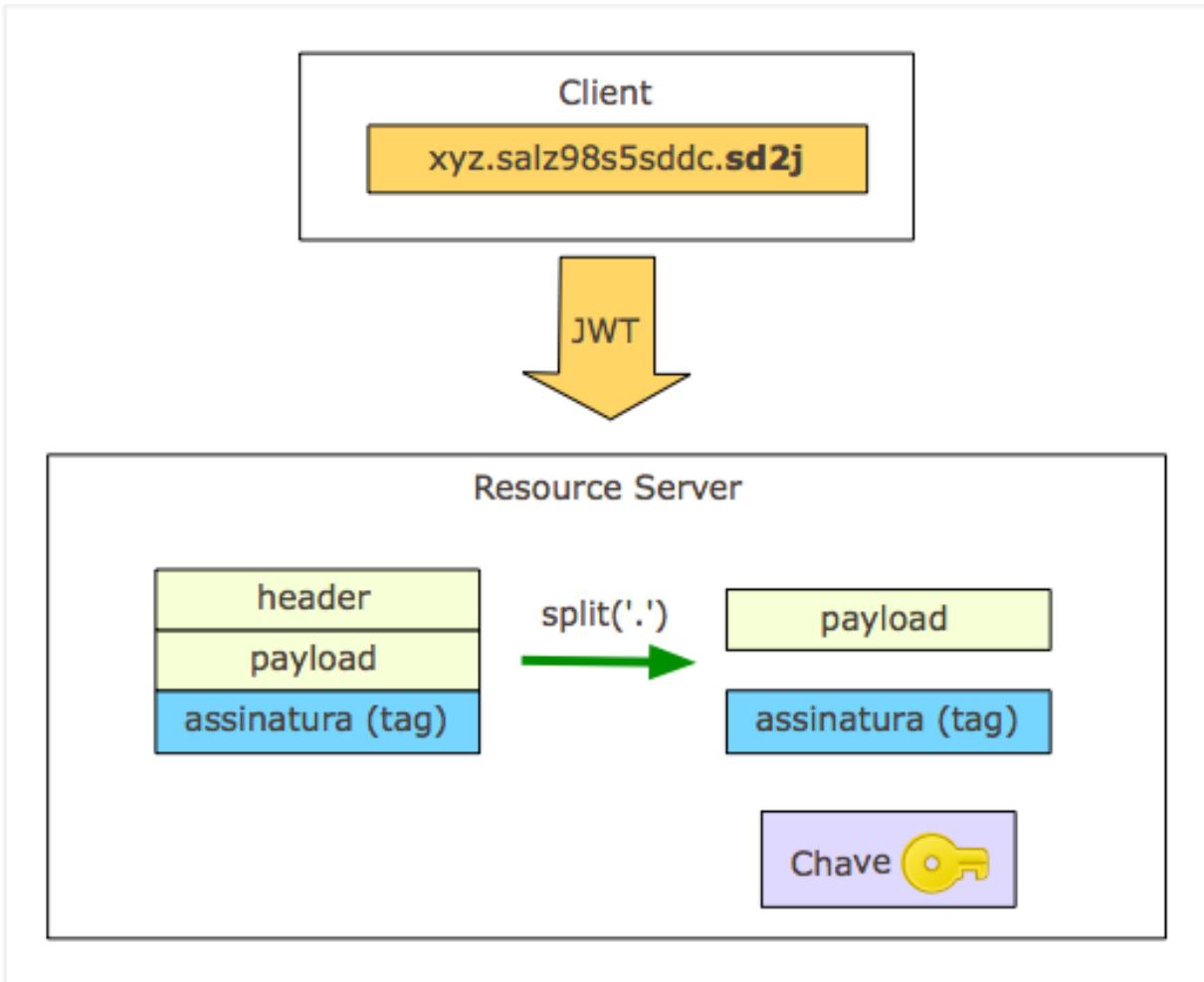


Figura 15.3: Validação do JWT token

Quando um Resource Server recebe um token durante o acesso a algum recurso protegido, ele precisa extraír cada segmento do token JWT para começar a validação. A partir do header extraído, o Resource Server pode identificar qual algoritmo foi utilizado para assinar o token de modo que seja possível aplicá-lo no processo de assinatura realizado pelo Authorization Server. Dessa forma, o Resource Server gera uma nova assinatura e compara com a que foi recebida na terceira seção do token.

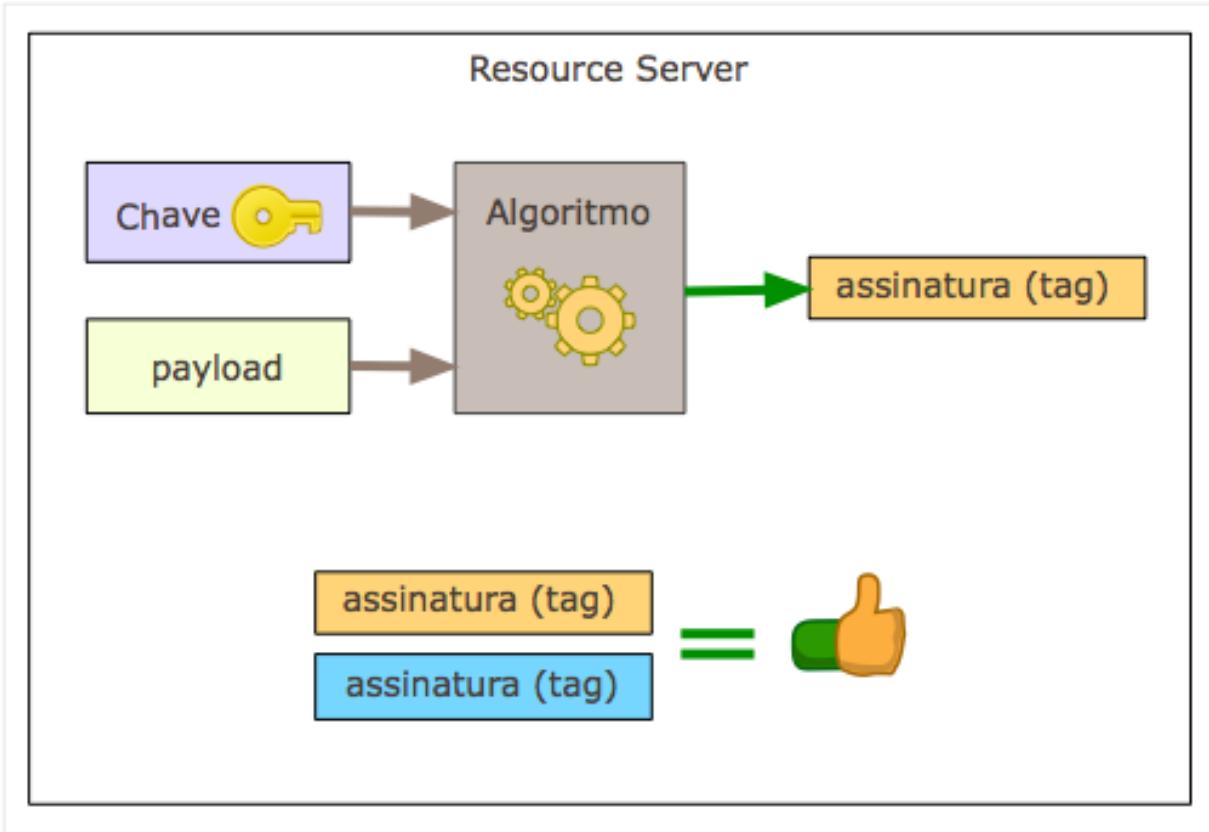


Figura 15.4: Verificação da assinatura JWT do token

Caso as assinaturas sejam iguais, o token pode ser considerado válido e os dados do *payload* podem ser usados sem perigo.

A questão agora é qual algoritmo utilizar? Os algoritmos possíveis tanto para assinatura quanto para criptografia estão definidos na especificação **JWA (JSON Web Algorithm)**. Os algoritmos definidos na especificação JWA podem ser usados para gerar dois tipos de assinatura ou criptografia: simétrica ou assimétrica.

Assinatura simétrica

Ao utilizar um algoritmo de assinatura simétrica, tanto a entidade que gera a assinatura quanto a que valida o conteúdo utilizam uma mesma chave compartilhada. No caso de um OAuth Provider, somente o Authorization Server e o Resource Server possuem acesso às chaves. Para assinar o conteúdo de maneira simétrica,

podemos usar funções de criptografia do tipo HMAC (*Hash Based Message Authentication Code*).

MAC e HMAC

A sigla MAC, cujo significado é *Message Authentication Code*, é uma porção de informação que pode ser usada para verificar a autenticidade de um conteúdo bem como a sua integridade.

Quando geramos uma assinatura, estamos criando exatamente um valor MAC. Quando falamos de HMAC, ou seja *Hash Based MAC*, estamos tratando de um tipo específico de MAC que é gerado a partir de uma função criptográfica de hash. Mais sobre esse conteúdo pode ser encontrado em

https://en.wikipedia.org/wiki/Hash-based_message_authentication_code.

Existem diversos exemplos de funções de criptografia do tipo HMAC e, dentre eles, podemos citar alguns famosos como MD5 e SHA256. Nas imagens anteriores nas quais mostrei como um token pode ser assinado e verificado, tanto o Authorization Server quanto o Resource Server compartilhavam de uma mesma chave para gerar a assinatura. Esse é um exemplo prático da utilização de assinatura simétrica.

Assinatura assimétrica

Quando usamos assinatura assimétrica, o Authorization Server detém um par de chaves, sendo uma pública e outra privada. A chave privada é usada para assinar o token JWT, enquanto que a chave pública é compartilhada com o Resource Server que deve utilizá-la para validar o conteúdo. Dessa forma, apenas o Authorization Server pode assinar um token JWT. Como exemplos de algoritmos de criptografia assimétrica, podemos citar RSA e Elliptic Curve.

Para conhecer um pouco sobre a estrutura dessas chaves e como elas podem ser representadas para o contexto do JWT, visite o link <https://mkjwk.org/>. Nele é possível gerar um par de chaves para assinatura assimétrica seguindo a especificação **JWK (JSON Web Key)**. O JWK nada mais é do que um JSON que define uma estrutura para representar chaves de criptografia de forma padronizada.

Apesar das vantagens de segurança obtidas ao utilizar um algoritmo de assinatura assimétrico, considere o custo de processamento envolvido. O algoritmo RSA é mais complexo, usa mais recursos de processamento e utiliza chaves maiores do que as usadas em algoritmos simétricos. Tudo isso implica no funcionamento da sua aplicação, podendo refletir até mesmo na experiência do usuário.

Criptografia do payload

A criptografia, diferentemente da assinatura, vai proteger contra a leitura não autorizada de um conteúdo, enquanto que a assinatura garante a integridade do conteúdo. Conforme já citado anteriormente, também temos uma especificação para geração de tokens com o payload criptografado. A padronização de como um token pode ser protegido através de criptografia é definida através da especificação JWE, que se trata de outro caso específico de JWT.

Conforme podemos ver na figura a seguir, a estrutura de um token JWE é um pouco diferente da que vimos para o JWS. Em vez de três seções separadas por ponto, agora temos cinco seções:

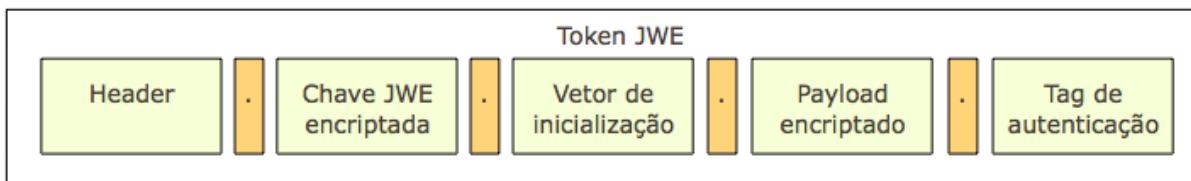


Figura 15.5: Estrutura de um Token JWE

Um token JWE ainda contém o segmento com um cabeçalho (*JOSE header*), entretanto a especificação JWE adiciona mais alguns atributos além dos que já vimos para o JWS. Para se ter uma ideia, o cabeçalho de um token JWE descreve dois tipos de algoritmos de criptografia: um algoritmo para proteção do payload e outro para criptografia da chave usada para criptografar o payload (essa chave compõe a segunda parte do token JWE).

A terceira parte do token JWE pode conter um vetor de inicialização que se trata de uma string gerada randomicamente para adicionar aleatoriedade aos dados criptografados. Em seguida, temos a quarta parte que se trata do payload, e a quinta e última que contém uma chave que permite a verificação da autenticidade do token.

Não entrarei em detalhes sobre como gerar um token JWE, nem sobre os algoritmos envolvidos, pois além de não ser esse o objetivo do livro, tudo isso poderia ser assunto para um capítulo totalmente dedicado (talvez até mesmo um outro livro). Entretanto, quando for necessário utilizar JWE, você pode contar com bibliotecas prontas para a realização dessa tarefa. Uma biblioteca bastante conhecida é a `nimbus-jose-jwt`, que pode ser melhor estudada através do link <https://connect2id.com/products/nimbus-jose-jwt>.

15.7 Usando JWT com Spring Security OAuth2

Como vamos testar o uso de JWT no projeto `bookserver`, não precisaremos mais usar os mecanismos de armazenamento de token que viemos utilizando até o momento, ou seja, `InMemoryTokenStore` e `JdbcTokenStore`. Precisamos agora utilizar um `JwtTokenStore` em nosso Authorization Server e também no Resource Server.

Entretanto, antes de começar a configuração que dará suporte ao JWT, importe o projeto `bookserver-jwt` do diretório `livro-spring-`

`oauth2/capitulo-15`, e adicione a seguinte dependência no arquivo `pom.xml` do projeto:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-jwt</artifactId>
</dependency>
```

Observe no pacote `br.com.casadocodigo.configuracao.seguranca.oauth` que as classes de configuração do Authorization Server e do Resource Server, que antes eram declaradas dentro de `ConfiguracaoOAuth2`, agora estão declaradas em arquivos separados. Como precisaremos escrever bastante código nessas classes, é melhor que estejam separadas, pois assim aumentaremos a legibilidade do código.

Suporte a JWT no Authorization Server

Vamos começar a configuração pelo Authorization Server definindo os beans necessários, para que a aplicação `bookserver-jwt` possa suportar tokens JWT. O primeiro bean a ser definido é um objeto do tipo `JwtAccessTokenConverter`, conforme mostrado a seguir:

```
@Configuration
@EnableAuthorizationServer
protected static class AuthorizationServerConfiguration
    extends AuthorizationServerConfigurerAdapter {
    // atributos omitidos
    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        converter.setSigningKey("assinatura-bookserver");
        return converter;
    }
    // resto do código omitido
}
```

A classe `JwtAccessTokenConverter` é responsável por traduzir tokens codificados com JWT para informações de autenticação OAuth, e

vice-versa. Para realizar o trabalho de conversão, essa classe conta com duas outras muito importantes: `Signer` e `SignatureVerifier`. Ambas as classes permitem assinar o payload de um token codificado com JWT e verificar a assinatura de um token, respectivamente.

No trecho de código anterior, estamos definindo manualmente uma chave pública para assinatura do payload JWT. Como o código apresentado está sendo usado apenas como exemplo, a chave pública está visível no código. Entretanto, em um projeto real, essa chave deveria ser carregada no código dinamicamente (por exemplo, por meio do uso de variáveis de ambiente).

O próximo `bean` a ser declarado é responsável por permitir a leitura de tokens codificados com JWT e trata-se de um `bean` do tipo `JwtTokenStore`. Apesar de implementar `TokenStore`, essa classe não fornece realmente um mecanismo de armazenamento, pois ela não realiza persistência dos dados associados a um token. Ao contrário disso, um `JwtTokenStore` permite a recuperação de dados associados a um token OAuth2 JWT.

Para que uma instância de `JwtTokenStore` possa realizar tais operações de tradução, ela depende do `bean` que já declaramos, responsável pela conversão de dados: `JwtAccessTokenConverter`. Veja como ficou a declaração do `bean` do tipo `JwtTokenStore`.

```
@Bean  
public JwtTokenStore jwtTokenStore() {  
    return new JwtTokenStore(accessTokenConverter());  
}
```

Agora que estamos definindo o `bean` `jwtTokenStore`, este já estará disponível no contexto do Spring quando a aplicação iniciar. Contudo, para que ele seja utilizado nos fluxos do OAuth 2.0, é necessário configurar um `bean` do tipo `DefaultTokenServices` para que ele passe a usar o `JwtTokenStore` que definimos.

```
@Bean
@Primary
public DefaultTokenServices tokenServices() {
    DefaultTokenServices tokenServices = new DefaultTokenServices();
    tokenServices.setTokenStore(jwtTokenStore());
    tokenServices.setSupportRefreshToken(true);

    return tokenServices;
}
```

Atenção, pois esse bean já é declarado automaticamente pelo Spring Security OAuth2. Por conta disso, foi necessário adicionar a anotação `@Primary` para que o Spring considere a nossa declaração com prioridade.

Agora que estamos declarando todos os beans necessários para suportar tokens JWT, precisamos fazer com que esses beans sejam usados pelo Authorization Server. Para isso, altere o código do método `configure(AuthorizationServerEndpointsConfigurer endpoints)` para que fique da seguinte maneira:

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints)
    throws Exception {

    DefaultOAuth2RequestFactory requestFactory
        = new DefaultOAuth2RequestFactory(clientDetailsService);

    requestFactory.setCheckUserScopes(true);

    endpoints
        .authenticationManager(authenticationManager)
        .requestFactory(requestFactory)
        .tokenStore(jwtTokenStore())
        .accessTokenConverter(accessTokenConverter());
}
```

Inicie a aplicação `bookserver-jwt` e tente obter um token de acesso utilizando o grant type **password**, conforme o exemplo de comando `CURL` mostrado a seguir.

```

curl -X POST -H "Authorization: Basic Y2xpZW50ZS1hcHA6MTIzNDU2" \
-H "Content-Type: application/x-www-form-urlencoded" \
-H "Accept: application/json" \
-d
'grant_type=password&username=oauth@mailinator.com&password=123&scope=read'
' "http://localhost:8080/oauth/token"

```

Após a execução do comando anterior, você deve obter na saída do seu terminal um conteúdo similar ao que recebi quando executei esse comando em minha máquina:

```

{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsiYm9va3MiXSwidXNlc19uYW11
Ijoib2F1dGhAbWFpbGluYXRvcizjb20iLCJzY29wZSI6WyJyZWFKIiwid3JpdGUiXSwiZXhwIj
oxNDkxODM1MzgxLCJhdXRob3JpdGllcyI6WyJST0xFX1VTVUFSSU9fQ09NVU0iLCJyZWFKIl0s
Imp0aSI6IjExM2EzzGMyLWE2ODItNDIwMS05NDg3LWUxYjcyOTk5NjczMiIsImNsawVudF9pZC
I6ImNsawVudGUTYXBwIn0.NU6v5kh6esTNNdKAGDSH98jfqeBFplhlC5kWGtwBpQY",
  "token_type": "bearer",
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsiYm9va3MiXSwidXNlc19uYW11
Ijoib2F1dGhAbWFpbGluYXRvcizjb20iLCJzY29wZSI6WyJyZWFKIiwid3JpdGUiXSwiYXRpIj
oiMTEzYTNmYzItYT4Mi00MjAxLTk00DctZTFiNzI50Tk2NzMyIiwiZXhwIjoxNDk0Mzg0MTgx
LCJhdXRob3JpdGllcyI6WyJST0xFX1VTVUFSSU9fQ09NVU0iLCJyZWFKIl0sImp0aSI6IjJlMW
ZkZjFhLTiyZjktNDYyMi04YjMwLTgwNTZmZWQ4ZWZkMCIsImNsawVudF9pZCI6ImNsawVudGUT
YXBwIn0.UVSRFw38hFGcDb2eaMeMLyS952AGibUqfxcz0ldjnB8",
  "expires_in": 43199,
  "scope": "read write",
  "jti": "113a3dc2-a682-4201-9487-e1b729996732"
}

```

Não se assuste com o tamanho da resposta. Como um token codificado com JWT possui cabeçalho, payload e assinatura, é normal que o token fique um pouco grande. Uma questão importante a ser observada é que, quanto mais *claims* forem adicionadas no payload, maior será o token (sendo isso um bom motivo para tomar cuidado com o que se escolhe para colocar no payload).

Com a configuração realizada para o Authorization Server, já podemos gerar tokens de acesso codificados com JWT. Entretanto, ainda não é possível validar o token caso tentarmos acessar o endpoint de consulta utilizando um token JWT.

Suporte a JWT no Resource Server

Agora vamos realizar uma configuração semelhante para o Resource Server. A configuração é semelhante, pois o Resource Server precisa validar o token recebido durante uma requisição para um recurso protegido. Para realizar a validação, já sabemos que é necessário utilizar um `JwtTokenStore` e um `JwtAccessTokenConverter`.

No caso da aplicação `bookserver-jwt`, como usamos os dados do usuário que permitiu a geração do token para buscar os seus livros, precisamos criar o `JwtAccessTokenConverter` de forma diferente adicionando como dependência um outro tipo de

`AccessTokenConverter`. Abra a classe `ResourceServerConfiguration` e adicione a seguinte definição de `bean`:

```
@Configuration
@EnableResourceServer
protected static class ResourceServerConfiguration
    extends ResourceServerConfigurerAdapter {

    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        converter.setVerifier(verifier());
        converter.setAccessTokenConverter(defaultAccessTokenConverter());
        converter.setSigningKey("assinatura-bookserver");
        return converter;
    }
    // resto do código omitido
}
```

O código que adicionamos ainda não compila, pois estamos usando dois métodos que não foram definidos. Vamos primeiro definir o

método `verifier` dentro da classe `ResourceServerConfiguration`, conforme mostrado a seguir:

```
@Bean  
public SignatureVerifier verifier() {  
    return new MacSigner("assinatura-bookserver");  
}
```

Através da classe `MacSigner` que estamos criando é que o *converter* poderá validar a assinatura de um token codificado com JWT. Repare que estamos informando qual a chave usada para assinar os tokens, pois a mesma chave será usada na validação. Lembre-se de que, para validar o token, o Resource Server precisa gerar uma assinatura igual a que foi gerada pelo Authorization Server.

O código ainda não compila, pois precisamos declarar o método `defaultAccessTokenConverter` na classe `ResourceServerConfiguration`, que será responsável por criar o bean conversor semelhante ao que criamos para o Authorization Server.

```
@Bean  
public AccessTokenConverter defaultAccessTokenConverter() {  
    DefaultAccessTokenConverter tokenConverter  
        = new DefaultAccessTokenConverter();  
    tokenConverter.setUserTokenConverter(userTokenConverter());  
    return tokenConverter;  
}
```

Temos outro problema agora. Esse método depende de outro método que ainda não definimos e que, por sinal, tem a responsabilidade de criar um `bean` do tipo `UserAuthenticationConverter`. É através desse `bean` que a aplicação será capaz de recuperar um objeto do tipo `UserDetails`, para em seguida configurar o objeto `Authentication`, mantido no contexto do Spring Security.

Lembre-se de que recuperamos um `Authentication` do contexto do Spring Security quando vamos buscar os livros de um Resource

Owner. Portanto, crie o método `userTokenConverter` na classe `ResourceServerConfiguration` conforme mostrado a seguir:

```
@Bean
public UserAuthenticationConverter userTokenConverter() {
    DefaultUserAuthenticationConverter converter
        = new DefaultUserAuthenticationConverter();
    converter.setUserDetailsService(userDetailsService);
    return converter;
}
```

O converter que acabamos de definir ainda precisa de um objeto do tipo `UserDetailsService`. Para isso, vamos injetar um bean do tipo `DadosDoUsuarioService` dentro da classe `ResourceServerConfiguration`. É através dessa classe que serão recuperados os dados do usuário que autorizou a geração do token.

```
@Autowired
private DadosDoUsuarioService userDetailsService;
```

Agora todo o código compila, mas ainda precisamos configurar o token store especializado em JWT. Adicione o seguinte trecho de código na classe `ResourceServerConfiguration` para configurar um `JwtTokenStore`.

```
@Bean
public TokenStore tokenStore() {
    JwtTokenStore store = new JwtTokenStore(accessTokenConverter());
    return store;
}
```

Assim como configuramos para o Authorization Server, também precisamos configurar apenas mais um bean muito importante. Adicione a declaração do bean `DefaultTokenServices` na classe `ResourceServerConfiguration`:

```
@Bean
@Primary
public DefaultTokenServices tokenServices() {
    DefaultTokenServices tokenServices = new DefaultTokenServices();
```

```
        tokenServices.setTokenStore(tokenStore());
    return tokenServices;
}
```

Ufa! Chega de declarar tantos beans. Agora é hora de fazer uso de algumas dessas declarações. Portanto, altere o seguinte método de configuração da classe `ResourceServerConfiguration` para referenciar o bean `tokenServices` e o `tokenStore` que declaramos anteriormente.

```
@Override
public void configure(ResourceServerSecurityConfigurer resources)
    throws Exception {
    resources.resourceId(RESOURCE_ID);
    resources.tokenServices(tokenServices());
    resources.tokenStore(tokenStore());
}
```

Pronto, agora inicie a aplicação novamente, solicite um novo token de acesso e tente acessar o endpoint de consulta de livros utilizando o token JWT recém-gerado, conforme mostrado no comando `cURL` de exemplo. Obviamente o token que você precisa utilizar deve ser diferente do qual eu estou usando.

```
curl -X GET -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOlsiYm9va3MiXSwidXNlc19uYW1lI
joib2F1dGhAbWFpbGluYXRvcj5jb20iLCJzY29wZSI6WyJyZWFKIiwid3JpdGUiXSwiZXhwIjo
xNDkxODM4MjIyLCJhdXRob3JpdGllcyI6WyJST0xFX1VTUFSSU9fQ09NVU0iLCJyZWFKI10sI
mp0aSI6IjRkMzk0ODI3LTA2YmMtNDFhMy1hMjdjLWU1MTg1NmMwN2YzMCIiImNsawVudF9pZCI
6ImNsawVudGUtYXBwIn0.vqmPrX7GIuvNvtjmjlVKIB7gyC0CfaGOTpSPawwEcZI"
"http://localhost:8080/api/v2/livros"
```

15.8 Adicionando claims customizadas

Os tokens codificados com JWT que geramos até agora contêm apenas *claims* padrão e algumas que o próprio Spring Security OAuth2 adiciona. Porém, como podemos fazer caso precisarmos de mais *claims* do usuário?

Em alguns projetos, pode ser necessário manter vários dados relacionados a um usuário no payload do token para evitar *round trips* para o Authorization Server. Como exemplo dessa situação, podemos citar uma aplicação em que o Authorization Server e o Resource Server são implementados de forma separada, de modo que apenas o Authorization Server tem acesso a todos os dados cadastrais de um usuário.

Para adicionar mais *claims* usando o Spring Security OAuth2, precisamos criar uma implementação de `TokenEnhancer` que nos permitirá aumentar os dados associados a um token antes que este seja armazenado, ou codificado no caso de um token JWT. Vamos agora criar a classe `DadosAdicionaisEnhancer` dentro do pacote `br.com.casadocodigo.configuracao.seguranca.oauth`, conforme mostrado a seguir:

```
@Component
public class DadosAdicionaisEnhancer implements TokenEnhancer {

    @Override
    public OAuth2AccessToken enhance(
        OAuth2AccessToken accessToken,
        OAuth2Authentication authentication) {

        Map<String, Object> additionalInformation = new HashMap<>();

        ResourceOwner usuario = (ResourceOwner)
authentication.getPrincipal();

        additionalInformation.put("nome_usuario",
            usuario.getUsuario().getNome());

        DefaultOAuth2AccessToken defaultAccessToken
            = (DefaultOAuth2AccessToken) accessToken;

        defaultAccessToken.setAdditionalInformation(additionalInformation);

        return defaultAccessToken;
    }
}
```

```
 }  
}
```

Agora injete esse componente na classe

AuthorizationServerConfiguration :

```
@Autowired  
private DadosAdicionaisEnhancer tokenEnhancer;
```

E em seguida, référencia nosso tokenEnhancer no método de configuração configure(AuthorizationServerEndpointsConfigurer endpoints) .

```
// códigos omitidos  
TokenEnhancerChain tokenEnhancerChain = new TokenEnhancerChain();  
tokenEnhancerChain.setTokenEnhancers(  
    Arrays.asList(tokenEnhancer, accessTokenConverter()));  
  
endpoints  
    .authenticationManager(authenticationManager)  
    .requestFactory(requestFactory)  
    .tokenStore(jwtTokenStore())  
    .tokenEnhancer(tokenEnhancerChain)  
    .accessTokenConverter(accessTokenConverter());
```

Solicite a geração de um novo token JWT. Caso você fizer a decodificação base 64 do payload do token, poderá observar um conteúdo semelhante ao mostrado a seguir. Repare no atributo nome_usuario que deve estar presente no conteúdo JSON do payload.

```
{  
    "nome_usuario": "José OAuth",  
    "aud": [  
        "books"  
    ],  
    "user_name": "oauth@mailinator.com",  
    "scope": [  
        "read",  
        "write"
```

```
],
  "exp": 1491840495,
  "authorities": [
    "ROLE_USUARIO_COMUM",
    "read"
  ],
  "jti": "bce3a760-b4e5-4aa9-87c3-ed4c934a0fe4",
  "client_id": "cliente-app"
}
```

15.9 Conclusão

Este longo capítulo apresentou conceitos importantes sobre diferentes estratégias para validação de token de acesso. Ao conhecer tais abordagens, você fica preparado para resolver problemas bem específicos na arquitetura de um projeto que utiliza OAuth 2.0 para proteger uma API. Entender como funciona cada mecanismo lhe permite avaliar as vantagens e desvantagens de cada estratégia.

Ao ler este capítulo, além de entender como funciona o mecanismo de introspecção de tokens de acesso, que é chave para a validação remota, você também fica conhecendo bastante sobre como trabalhar com tokens JWT. Diferentemente de apenas usar o framework, você conhece a estrutura completa de um JWT e como funcionam os mecanismos de criptografia e de assinatura. Com todos os conceitos apresentados, tenho certeza de que será muito mais fácil usar o framework nos cenários mais distintos possíveis.

No próximo capítulo, estudaremos mais alguns conceitos importantes que vão diferenciar seu conhecimento a respeito de OAuth 2.0. Vamos entender de uma vez por todas por que o OAuth 2.0 não é um protocolo de autenticação.

CAPÍTULO 16

OAuth 2.0 como base para autenticação

Dentre muitos conceitos que estudamos até agora sobre o protocolo OAuth 2.0, temos bem clara a definição de que ele *não se trata de um protocolo de autenticação*. É muito importante ter consciência disso para evitar problemas de segurança. Em razão do fluxo de autorização e solicitação de access tokens contarem com a autenticação do Resource Owner e do Client respectivamente, muitos desenvolvedores são levados a interpretar o protocolo OAuth 2.0 como uma solução para autenticação.

Outra situação que acaba gerando bastante confusão acontece quando queremos permitir que um usuário accesse nossa aplicação usando sua conta do Google ou do Facebook. Pelo fato desse tipo de serviço de autenticação contar com fluxos do OAuth 2.0, muitos desenvolvedores acabam misturando os conceitos.

Para se ter uma ideia do problema, nem o Client e nem o Resource Server têm condições de identificar se um usuário foi autenticado. O Client não tem esta capacidade porque, através do token de acesso, não é possível obter dados do usuário que autorizou a geração do token (lembre-se, o token de acesso é **opaco** para o Client).

O Resource Server, apesar de ser capaz de descobrir os dados do usuário que autorizou a geração do token (seja acessando o banco de dados ou usando introspecção de tokens ou JWT), não pode identificar se o usuário que autorizou a geração do token está presente durante o acesso aos recursos.

16.1 Por que não usar OAuth 2.0 para autenticação?

Justin Richter e Antonio Sanso citam no livro *OAuth2 in Action* vários problemas que podem ser encontrados quando se utiliza OAuth 2.0 para autenticação. Um deles é considerar um access token como prova de que o usuário está autenticado. O problema nisso é que o token sendo usado pode nem mesmo ter sido obtido com a intervenção do usuário.

Um exemplo clássico desse tipo de situação é quando obtemos um token de acesso através de refresh tokens. Aliás, um dos objetivos de se utilizar refresh tokens é poder obter um novo token de acesso sem precisar que o usuário esteja presente para realizar a autorização. Além da questão de o usuário estar ou não presente, como a **audiência** do token não é definida para o Client, ele não pode inferir nada a partir do token.

Considerando que um Client não está protegido contra injeção de token de acesso, os resultados de usar OAuth 2.0 como autenticação podem ser bem ruins. Imagine uma situação na qual um Client que usa o grant type Implicit acabe recebendo um token de acesso que não foi autorizado pelo usuário que está presente. Se o Client não estiver contando com o parâmetro `state`, por exemplo, esse tipo de situação pode acontecer facilmente (veremos mais sobre problemas de segurança mais adiante). Nesse caso, o Resource Server acabaria aceitando tal token de acesso, pois ele não tem condições de identificar se o token foi enviado pelo verdadeiro dono.

Mas nem tudo está perdido. Devido à extensibilidade do protocolo OAuth 2.0, este pode ser usado como base para a criação de outros protocolos. Podemos usar o protocolo OAuth 2.0 para que o usuário delegue acesso à sua identidade em vez de acesso aos seus recursos. Nesse caso, um usuário pode manter recursos em uma aplicação X e sua identidade pode ser gerenciada por uma aplicação Y, conforme mostrado na próxima figura.

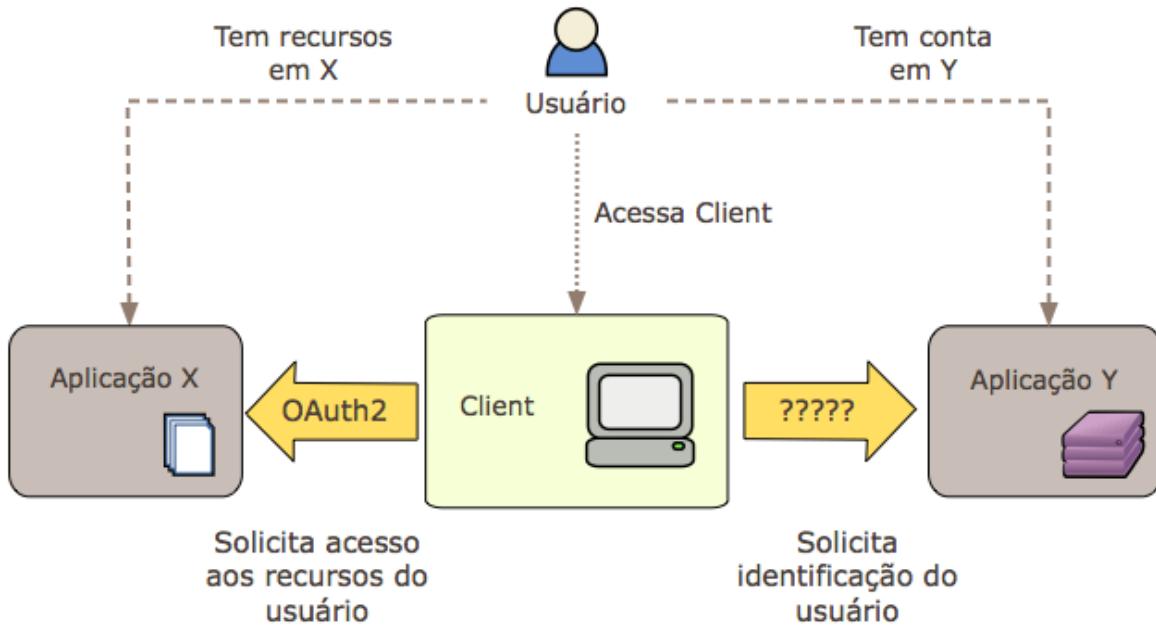


Figura 16.1: Utilizando autorização e autenticação

Agora temos as responsabilidades de autorização e autenticação bem separadas. Acessar recursos do usuário em uma aplicação X já é algo que estudamos bastante e já sabemos como fazer. Mas como usar outra aplicação para gerenciar a identidade e autenticação de um usuário?

Quando usamos OAuth 2.0, temos um token de acesso que serve como artefato de autorização aos recursos do usuário. Que tipo de artefato podemos usar para identificar um usuário? OpenID Connect vem para nos ajudar com todas essas questões.

16.2 OpenID Connect como solução para autenticação

A especificação OpenID Connect vem justamente como solução para o problema de autenticação. Diversas tentativas foram

realizadas por empresas diferentes até chegar no OpenID Connect, culminando em um padrão aberto que permite interoperabilidade entre aplicações e provedores de identidade. Entenda como **provedor de identidade** (*Identity Provider*) o componente que tem como responsabilidade autenticar um usuário retornando para o Client os dados de identificação. No contexto de autenticação, o Client também é chamado de *Relying Party*.

O Identity Provider – que, por sua vez, deve implementar a especificação OpenID Connect – é responsável por devolver para o Client o artefato que identifica o usuário como válido e presente. Este artefato, chamado de **ID Token**, se trata simplesmente de um token JWT que contém no payload os dados que permitem que o Client possa identificar o usuário. Veja a seguir as principais *claims* retornadas no JWT `id_token`.

Claim	Obrigatório	Descrição
<code>iss</code>	Sim	Identifica através de uma URL quem gerou o ID Token.

Claim	Obrigatório	Descrição
<code>sub</code>	Sim	Um identificador para o usuário autenticado.
<code>aud</code>	Sim	Audiência do ID Token. Deve ser o <code>client_id</code> da aplicação que está usando a autenticação via OpenID Connect.
<code>exp</code>	Sim	Indica, em segundos, quando o token expira. O valor em segundos é representado em horário no formato Unix.

Claim	Obrigatório	Descrição
<i>iat</i>	Sim	Indica, em segundos, quando o ID Token foi gerado. O valor em segundos é representado em horário no formato Unix.
<i>auth_time</i>	não	Momento em que o usuário se autenticou no Identity Provider.
<i>nonce</i>	Não	Valor associado ao Client que pode ser usado para mitigar um <i>replay attack</i> .
<i>acr</i>	Não	Esse atributo indica qual o mecanismo usado para autenticar o usuário.
<i>azp</i>	Não	Indica a parte autorizada, ou seja, o Client que foi autorizado. Nesse caso, também pode ser apresentado o <i>client_id</i> .

Além do **ID Token**, o Identity Provider também devolve um `access_token` que pode ser usado para buscar mais dados a respeito do usuário, como nome, endereço, e-mail, telefone e outros atributos que são definidos na seção *Standard Claims* da especificação OpenID Connect Core. Esta pode ser acessada através do link http://openid.net/specs/openid-connect-core-1_0.html#StandardClaims.

Você vai perceber que o protocolo OpenID Connect possui várias especificações complementares, tornando o processo de entendimento um pouco complicado. Entretanto, para o desenvolvedor que precisa implementar um Client, a especificação *OpenID Connect Basic Client Implementer's Guide* pode ajudar bastante por ser mais simples e direta. Esta especificação pode ser acessada em http://openid.net/specs/openid-connect-basic-1_0.html.

16.3 Fluxo de autenticação usando OpenID Connect

Agora que já estamos munidos de informação para entender **o que** é necessário para autenticar um usuário usando o padrão OpenID Connect, vamos entender **como** realizar a autenticação respondendo às seguintes perguntas:

- Como o Client obtém um usuário autenticado por outra entidade?
- Como o usuário se autentica e onde?
- Como associar um usuário autenticado por outra aplicação aos recursos criados pelo próprio usuário na aplicação cliente?

Para responder essas perguntas, observe na próxima figura um cenário de autenticação utilizando OpenID Connect.

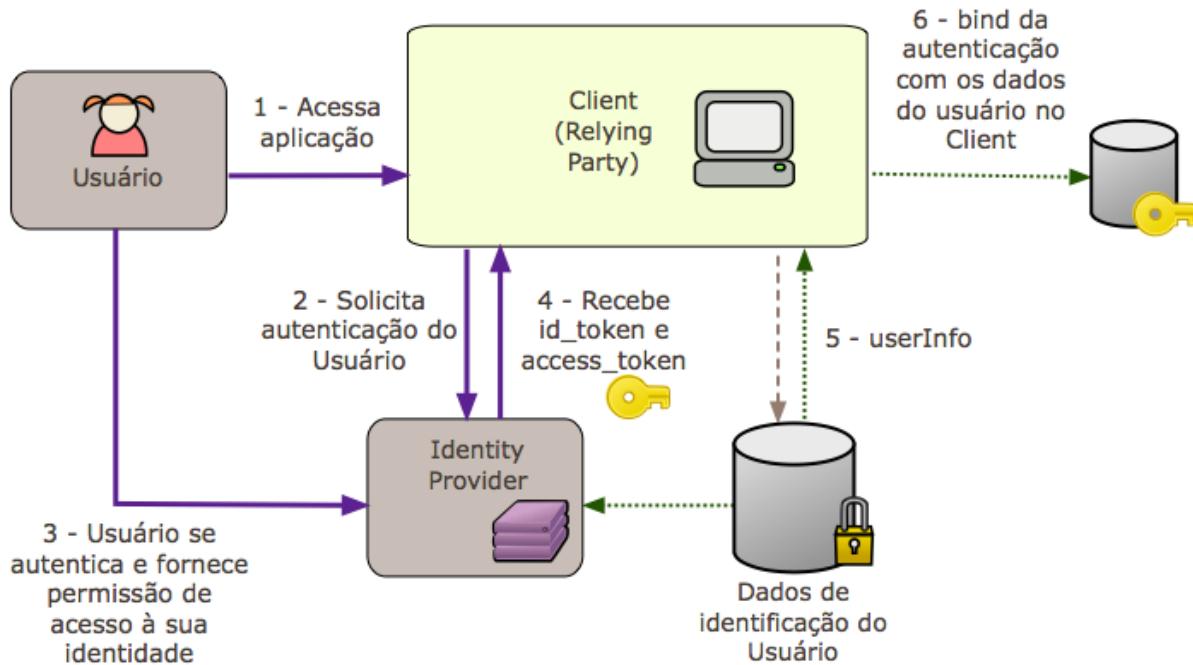


Figura 16.2: Fluxo de autenticação usando OpenID Connect

No fluxo apresentado, note que as quatro primeiras etapas seguem exatamente o mesmo fluxo definido pelo grant type Implicit do

OAuth 2.0. Isso acontece justamente por conta do OpenID Connect ser construído usando o OAuth 2.0 como base.

Embora a figura apresente o grant type Implicit, também podemos usar o grant type Authorization Code. Independente do grant type utilizado, é na etapa de solicitação do token que recebemos os campos `access_token` e `id_token`. Perceba que deve existir uma relação de confiança entre o Relying Party e o Identity Provider para que se tenha o mínimo de garantia de que o usuário autenticado realmente é válido.

Além dos dados obtidos a partir do `id_token`, o Client também pode solicitar mais informações sobre o usuário, conforme falei anteriormente. Esses dados podem ser obtidos através de um endpoint especial chamado `userInfo`, que deve ser fornecido por qualquer servidor que implemente a especificação OpenID Connect. Para saber mais detalhes sobre o endpoint `userInfo`, acesse http://openid.net/specs/openid-connect-core-1_0.html#UserInfo.

Agora imagine que estamos usando OpenID Connect para autenticar os usuários da aplicação `bookserver`. Estes usuários querem usar o sistema para cadastrar e avaliar os livros que já leram. Como manter uma associação entre os dados de autenticação obtidos do Identity Provider com os dados gerados pelo próprio usuário em nossa aplicação? Para isso, podemos ter uma entidade que represente o usuário em nosso sistema de uma forma parecida com a que já fazemos para associar um usuário aos livros criados por ele.

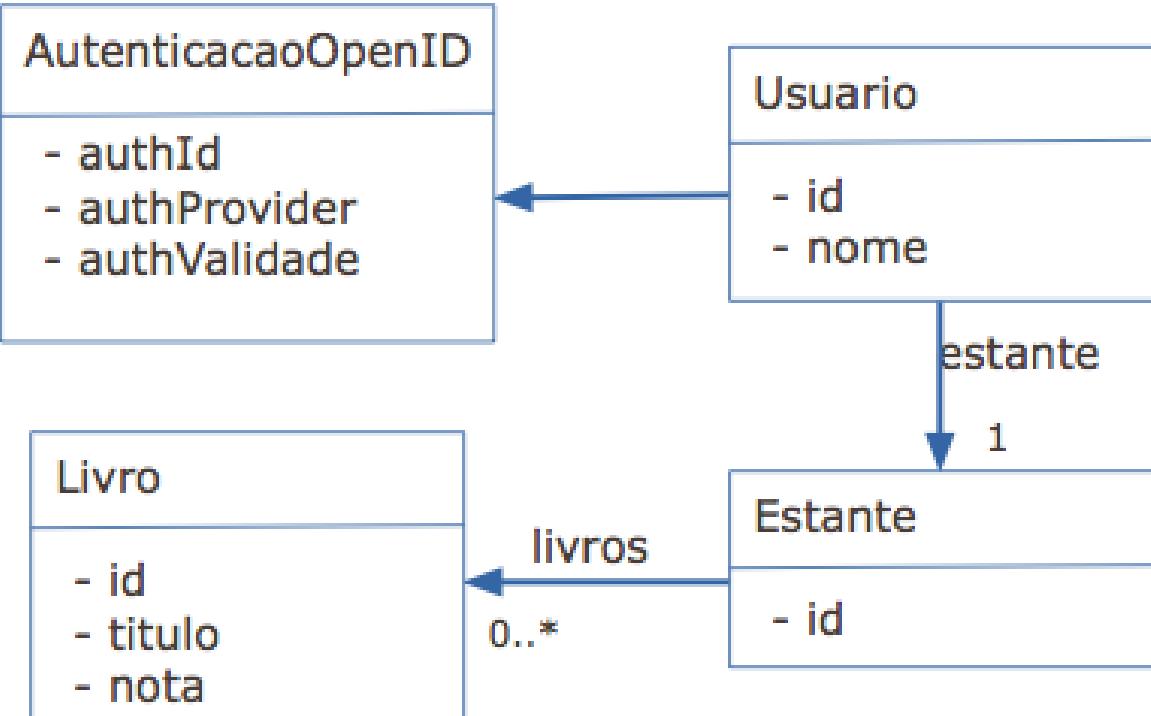


Figura 16.3: Bind dos dados de autenticação com o usuário e seus livros

A diferença é que agora também podemos manter alguns atributos obtidos a partir do JWT `id_token`. Podemos guardar o atributo `sub` que identifica o usuário no Identity Provider. Além disso, para garantir a unicidade do registro, também é indicado mantermos no banco de dados a identificação do Provider, ou seja, o valor recebido na claim `iss`.

E como sabemos a hora de solicitar a autenticação do usuário novamente? Para isso, podemos salvar no banco de dados a data de validade da autenticação do usuário (que geralmente deve ser bem curta e é devolvida pelo servidor pela claim `exp`). Os atributos `sub`, `iss` e `exp` foram mapeados na entidade `AutenticacaoOpenID` mostrada na imagem anterior através dos campos `authId`, `authProvider` e `authValidade`, respectivamente.

Agora você pode se perguntar: mas e se o usuário se autenticar novamente no Identity Provider, este não vai retornar um novo

código de identificação na `claim sub`? Para nossa tranquilidade, esse atributo sempre será o mesmo, pois se trata de um código que identifica o usuário no próprio Identity Provider. Se estivermos usando o Google como Identity Provider, uma `sub` representará um usuário do Google.

16.4 Interações dinâmicas entre Relying Party e Identity Provider

Para que um Relying Party possa usar os serviços oferecidos por um Identity Provider, ele precisa saber onde o provider está localizado. Pensando em uma aplicação que permite autenticação de usuários através do serviço de OpenID Connect oferecido pelo Google, já sabemos de antemão qual o endereço do provider. Ler a própria documentação do Google OpenID Connect já é suficiente para descobrir qual o endereço do Identity Provider.

Entretanto, a especificação *OpenID Connect Discovery 1.0* (disponível em https://openid.net/specs/openid-connect-discovery-1_0.html) define maneiras de se obter o endereço do Identity Provider de um usuário de forma dinâmica. A maneira apresentada pela especificação é fornecer um endpoint `WebFinger` que pode ser acessado com parâmetros inferidos por meio de dados do usuário sendo autenticado (por exemplo, o e-mail do usuário).

O QUE É WEBFINGER?

Webfinger se trata de uma especificação criada com o objetivo de permitir a implementação de servidores que forneçam serviço de discovery de informações sobre pessoas ou entidades que possam ser identificadas por uma URI (usando o protocolo HTTP para isso). Em se tratando de OpenID Connect, a ideia de usar um serviço desse tipo serve para obter o endereço de um OpenID Provider em que determinado usuário tenha registro. Para mais detalhes sobre essa especificação, você pode dar uma olhada no link <https://tools.ietf.org/html/rfc7033>.

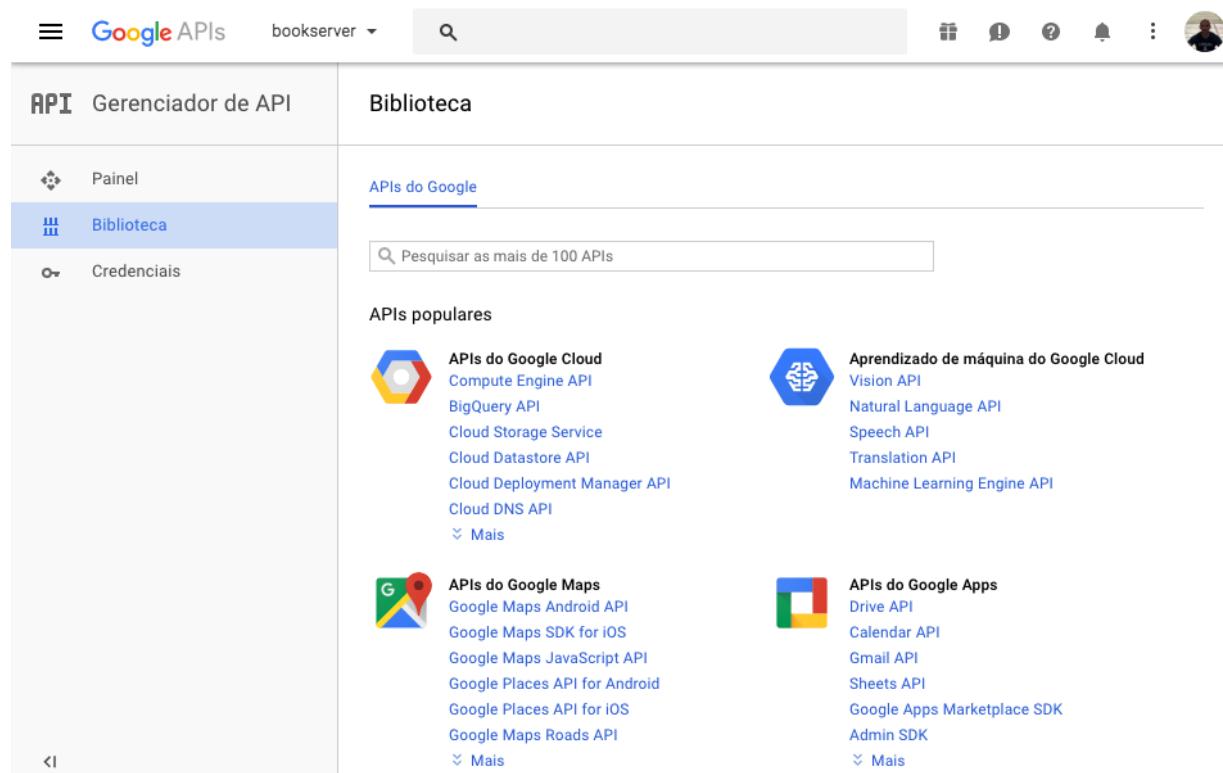
Após descobrir um Identity Provider, o Client pode obter informações essenciais sobre o Provider e sobre os endpoints OAuth 2.0 através de um documento JSON, disponível em um local conhecido pelo Client. Esse documento JSON é chamado **discovery document**.

Além de ser possível obter informações do servidor de autenticação de forma dinâmica, o servidor também pode obter informações do Client através do mecanismo de registro dinâmico. O Client pode se registrar automaticamente no Server usando os mecanismos definidos pela extensão OAuth 2.0 para *registro dinâmico de Client*.

16.5 Autenticação com uma conta Google

Neste livro, não vamos nos aprofundar muito sobre OpenID Connect, de modo que veremos na prática apenas a implementação do Client (ou Relying Party). Em vez de criar um Identity Provider, contaremos com um que pode ser considerado bem confiável: o Google. Além disso, a aplicação bookserver agora assumirá o papel de Client em relação ao Google, ou seja, os usuários dessa aplicação vão se autenticar usando uma conta Google.

Para que nossa aplicação possa ser reconhecida pelo Google, antes de mais nada precisamos registrar essa nossa aplicação (não vamos usar registro dinâmico). Acesse então a URL <https://console.developers.google.com>, e você verá um painel com APIs fornecidas pelo Google conforme mostrado a seguir.



The screenshot shows the Google APIs console interface. The left sidebar has three main items: 'Painel' (selected), 'Biblioteca' (selected), and 'Credenciais'. The 'Biblioteca' section is titled 'APIs do Google' and includes a search bar. Below it, there's a section for 'APIs populares' divided into four categories: 'APIs do Google Cloud' (Compute Engine API, BigQuery API, Cloud Storage Service, Cloud Datastore API, Cloud Deployment Manager API, Cloud DNS API, 'Mais'), 'Aprendizado de máquina do Google Cloud' (Vision API, Natural Language API, Speech API, Translation API, Machine Learning Engine API), 'APIs do Google Maps' (Google Maps Android API, Google Maps SDK for iOS, Google Maps JavaScript API, Google Places API for Android, Google Places API for iOS, Google Maps Roads API, 'Mais'), and 'APIs do Google Apps' (Drive API, Calendar API, Gmail API, Sheets API, Google Apps Marketplace SDK, Admin SDK, 'Mais').

Figura 16.4: Painel gerenciador de APIs do Google

O registro da aplicação é feito através da criação de um aplicativo no painel do desenvolvedor fornecido pelo Google. No meu caso, criei um aplicativo com o nome `bookserver`. Ao criar um aplicativo no Google, você deve solicitar a geração de um par de chaves (`client_id` e `client_secret`) através da opção `Criar Credenciais`. Ao clicar em `Criar Credenciais`, você precisará escolher a opção `ID do cliente OAuth`, conforme mostrado na lista de opções a seguir.

Credenciais

The screenshot shows the 'Credenciais' (Credentials) page in the Google Cloud Platform. At the top, there are three tabs: 'Credenciais' (selected), 'Tela de consentimento OAuth', and 'Confirmação de domínio'. Below the tabs, there are two buttons: 'Criar credenciais' (Create credential) with a dropdown arrow and 'Excluir' (Delete). A sidebar on the right shows a list of credentials with the identifier 'c26kmkl'. The main content area displays four options:

- Chave de API**: Identifica seu projeto usando uma chave de API simples para verificar cota e acesso.
- ID do cliente OAuth**: Solicita a permissão do usuário para que o seu aplicativo possa acessar os dados do usuário.
- Chave da conta de serviço**: Usa contas robô para ativar a autenticação do nível do app de servidor para servidor.
- Ajude-me a escolher**: Faz algumas perguntas para ajudar você a decidir que tipo de credencial usar.

Figura 16.5: Opções de criação de credenciais no Google

Na próxima tela, vamos definir o tipo de aplicativo como `Aplicativo da Web`. Ao selecionar essa opção, precisamos definir o nome da aplicação e a URL de redirecionamento que será usada para que o Google possa entregar o ID Token.

Criar ID do cliente

Tipo de aplicativo

- Aplicativo da Web
- Android [Saiba mais](#)
- Aplicativo do Google Chrome [Saiba mais](#)
- iOS [Saiba mais](#)
- PlayStation 4
- Outro

Nome

bookserver (cadastro de livros)

Restrições

Inserir origens do JavaScript, URIs de redirecionamento ou ambos

Origens JavaScript autorizadas

Para uso com solicitações de um navegador. Este é o URI de origem de um aplicativo cliente. Ele não pode conter um caractere curinga (`http://*.example.com`) ou um caminho (`http://example.com/subdir`). Se você usa uma porta não padrão, deve incluí-la no URI original.

`http://www.example.com`

URIs de redirecionamento autorizados

Para uso com solicitações de um servidor da Web. Este é o caminho em seu app ao qual os usuários são direcionados depois de autenticarem com o Google. O caminho será anexado com o código de autorização para acesso. É necessário ter um protocolo. Não pode conter fragmentos de URL ou caminhos relativos. Não pode ser um endereço IP público.

`http://localhost:8080/google/callback` ×

`http://www.example.com/oauth2callback`

Criar

[Cancelar](#)

Figura 16.6: Selecionando o tipo de aplicativo

Ao clicar em `Criar`, as credenciais serão geradas e apresentadas conforme vemos na figura a seguir.

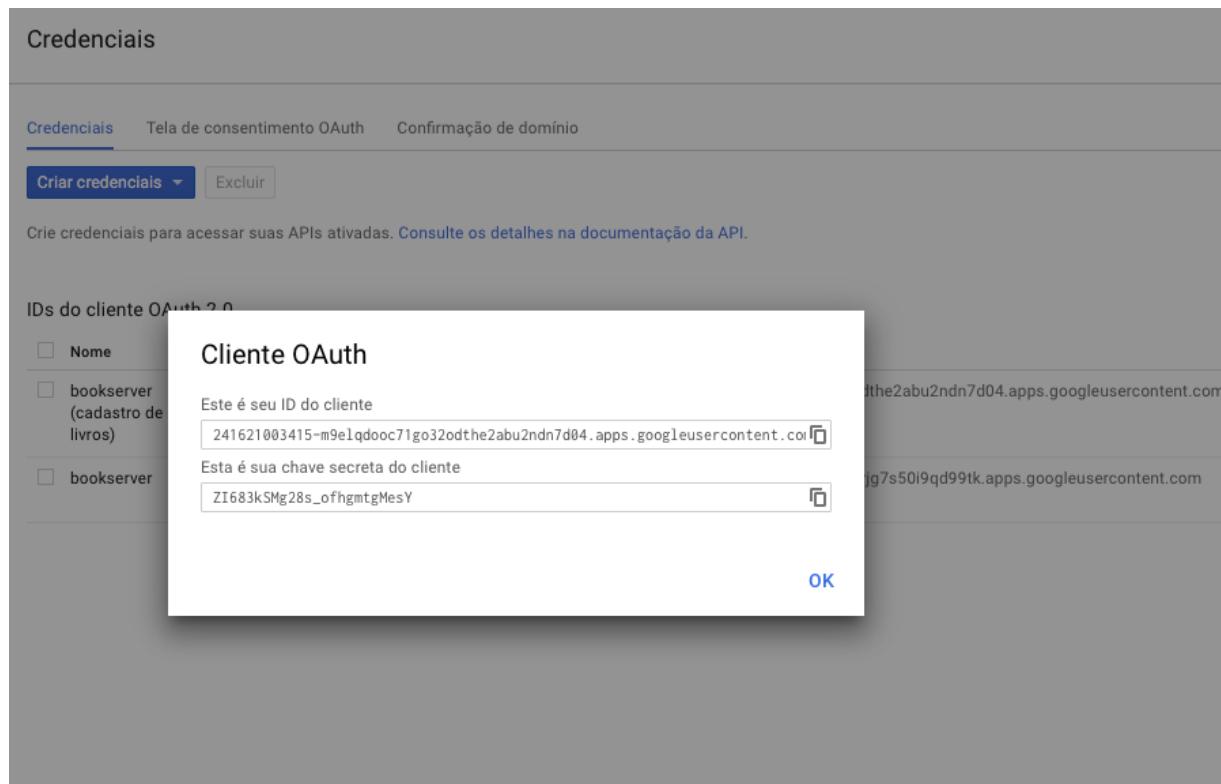


Figura 16.7: Credencias do aplicativo

É através das credenciais que sua aplicação pode fazer uso das APIs fornecidas pelo Google. Com o Client registrado no Identity Provider, já podemos começar a configurar a aplicação bookserver para utilizar o mecanismo de autenticação via OpenID Connect do Google.

Importe o projeto bookserver que se encontra no diretório `livro-spring-oauth2/capitulo-16`. Ao abri-lo, perceba que já não estamos mais implementando nenhuma funcionalidade de autenticação do usuário, ou seja, não existem mais `htmls` para login nem registro do usuário (tudo isso será fornecido pelo Google Open ID Connect).

Como a autenticação será realizada através do Google, não precisamos mais do campo `senha` na tabela `usuario`; entretanto, não é necessário removê-la. Contudo, precisamos criar a tabela `autenticacao_openid` para manter os dados do usuário que a aplicação bookserver vai receber do Identity Provider.

Para isso, acesse o banco de dados `bookserver` e execute a sequência de comandos SQL mostrada a seguir no console do MySQL (esses comandos também estão disponíveis no arquivo `database-openid.sql`):

```
create table autenticacao_openid (
    usuario_id int references usuario,
    authn_id varchar(255),
    authn_provider varchar(255),
    authn_validade datetime
);
```

Se preferir, limpe os registros existentes nas tabelas atuais usando os comandos a seguir:

```
delete from livro;
delete from estante;
delete from usuario;
```

Como a aplicação `bookserver` agora é Client do Google, então podemos usar a classe `OAuth2RestTemplate`, que já estudamos anteriormente, para obter os tokens de acesso e de identificação. Já sabemos que a especificação OpenID Connect permite o uso dos grant types Implicit e Authorization Code. Vamos usar o grant type Authorization Code por se tratar de um fluxo mais seguro e mais comum.

Agora, para realizar as configurações do Identity Provider que a aplicação `bookserver` vai acessar, precisamos declarar um `bean` do tipo `AuthorizationCodeResourceDetails`, assim como fizemos quando configuramos um Client OAuth com o Spring Security OAuth2. Abra a classe `GoogleOpenIdConnectConfig` que a essa altura está vazia e declare o `bean` mostrado a seguir.

```
@Configuration
@EnableOAuth2Client
public class GoogleOpenIdConnectConfig {
    @Bean
    public OAuth2ProtectedResourceDetails protectedResourceDetails() {
```

```

        AuthorizationCodeResourceDetails details = new
AuthorizationCodeResourceDetails();

        details.setClientId("client_id_da_sua_aplicacao");
        details.setClientSecret("client_secret_da_sua_aplicacao");
        details.setAccessTokenUri(
            "https://www.googleapis.com/oauth2/v4/token");
        details.setUserAuthorizationUri(
            "https://accounts.google.com/o/oauth2/v2/auth");
        details.setPreEstablishedRedirectUri(
            "http://localhost:8080/google/callback");
        details.setScope(Arrays.asList("openid", "email", "profile"));

        details.setUseCurrentUri(false);
        return details;
    }
}

```

O `client_id` e o `client_secret` são atributos que recuperamos durante o registro da aplicação. Mas e quanto aos outros atributos que estamos configurando? De onde vieram essas URLs?

Essas URLs estão disponíveis no **discovery document**, disponibilizado pelo Google OpenID Connect através do link <https://accounts.google.com/.well-known/openid-configuration>. Se você acessar o link, reparará que este documento fornece muito mais dados do que URLs de autorização e solicitação de token. Além disso, também podemos encontrar informações sobre como o `id_token` é assinado e muito mais.

Mais adiante, vamos precisar usar alguns dos dados que foram utilizados para configurar o objeto `details`, então para evitar duplicação de código, moveremos esses dados para o arquivo de configuração `application.yml`. Repare que também estamos definindo uma propriedade `userinfo_endpoint`, pois precisaremos dessa URL posteriormente para poder buscar dados do usuário após o processo de autenticação via OpenID Connect.

```
google:  
    client_id: client_id_da_sua_aplicacao  
    client_secret: client_secret_da_sua_aplicacao  
    access_token_uri: https://www.googleapis.com/oauth2/v4/token  
    user_authorization_uri: https://accounts.google.com/o/oauth2/v2/auth  
    redirect_uri: http://localhost:8080/google/callback  
    userinfo_endpoint: https://www.googleapis.com/oauth2/v3 userinfo
```

Agora, como vamos usar esses dados na classe

GoogleOpenIdConnectConfig ? Primeiramente criaremos uma outra classe que será usada para mapear as propriedades que definimos no arquivo application.yml . Crie a classe DiscoveryDocument dentro do pacote br.com.casadocodigo.configuracao.seguranca.openid , com o seguinte conteúdo:

```
@Component  
public class DiscoveryDocument {  
  
    @Getter  
    @Value("${google.client_id}")  
    private String clientId;  
  
    @Getter  
    @Value("${google.client_secret}")  
    private String clientSecret;  
  
    @Getter  
    @Value("${google.access_token_uri}")  
    private String accessTokenUri;  
  
    @Getter  
    @Value("${google.user_authorization_uri}")  
    private String userAuthorizationUri;  
  
    @Getter  
    @Value("${google.redirect_uri}")  
    private String redirectUri;  
  
    @Getter  
    @Value("${google.userinfo_endpoint}")
```

```
    private String userInfoEndpoint;  
  
}
```

Dessa forma, em vez de usar valores fixos na classe de configuração `GoogleOpenIdConnectConfig`, podemos injetar uma instância de `DiscoveryDocument` e configurar o objeto `details` de forma dinâmica, conforme mostrado adiante.

```
@Configuration  
@EnableOAuth2Client  
public class GoogleOpenIdConnectConfig {  
  
    @Autowired  
    private DiscoveryDocument discovery;  
  
    @Bean  
    public OAuth2ProtectedResourceDetails protectedResourceDetails() {  
        AuthorizationCodeResourceDetails details = new  
        AuthorizationCodeResourceDetails();  
  
        details.setClientId(discovery.getClientId());  
        details.setClientSecret(discovery.getClientSecret());  
        details.setAccessTokenUri(discovery.getAccessTokenUri());  
  
        details.setUserAuthorizationUri(discovery.getUserAuthorizationUri());  
        details.setPreEstablishedRedirectUri(discovery.getRedirectUri());  
        details.setScope(Arrays.asList("openid", "email", "profile"));  
  
        details.setUseCurrentUri(false);  
        return details;  
    }  
}
```

Parece que agora temos um código mais limpo, não é mesmo? Mas essa configuração ainda não é suficiente, pois precisamos configurar um bean `OAuth2RestTemplate` que vamos usar depois para obter os tokens de autenticação e de acesso. Adicione então o método a seguir na classe `GoogleOpenIdConnectConfig`:

```

@Bean
public OAuth2RestTemplate googleOpenIdRestTemplate(OAuth2ClientContext
clientContext) {
    OAuth2RestTemplate template = new OAuth2RestTemplate(
        protectedResourceDetails(), clientContext);

    template.setAccessTokenProvider(getAccessTokenProvider());
    return template;
}

private AccessTokenProviderChain getAccessTokenProvider() {
    AccessTokenProviderChain provider = new AccessTokenProviderChain(
        Arrays.asList(new AuthorizationCodeAccessTokenProvider()));
    return provider;
}

```

Logo a seguir, precisaremos criar a classe `OpenIdTokenServices` que tem como objetivo, salvar os dados de identificação do usuário assim que o mesmo for autenticado pelo *Identity Provider*. Crie a classe `OpenIdTokenServices` dentro do pacote `br.com.casadocodigo.configuracao.seguranca.openid` com o seguinte conteúdo:

```

@Service
@Transactional
public class OpenIdTokenServices {
}

```

Como as *claims* são definidas como `JSON`, precisaremos de um objeto que saiba fazer o mapeamento do `JSON` para a classe `TokenIdClaims` que vamos usar para representar os dados de identificação do usuário. Além disso, também vamos precisar usar a classe `RepositorioDeUsuarios` para persistir ou alterar um usuário já existente no banco de dados. Adicione os seguintes atributos na classe `OpenIdTokenServices`. Ao importar a classe `ObjectMapper`, utilize a que estiver no pacote `com.fasterxml.jackson.databind`.

```

@.Autowired
private RepositorioDeUsuarios repositorioDeUsuarios;

```

```
@Autowired  
private ObjectMapper jsonMapper;
```

Ainda precisamos criar a classe que vai conter as *claims* que vamos obter a partir do `id_token`. Crie então a classe `TokenIdClaims` no pacote `br.com.casadocodigo.configuracao.seguranca.openid`:

```
import com.fasterxml.jackson.annotation.JsonProperty;  
import lombok.Getter;  
import lombok.Setter;  
  
public class TokenIdClaims {  
  
    @Getter @Setter  
    @JsonProperty("azp")  
    private String authorizedParty;  
  
    @Getter @Setter  
    @JsonProperty("aud")  
    private String audience;  
  
    @Getter @Setter  
    @JsonProperty("sub")  
    private String subjectIdentifier;  
  
    @Getter @Setter  
    @JsonProperty("email")  
    private String email;  
  
    @Getter @Setter  
    @JsonProperty("at_hash")  
    private String accessTokenHashValue;  
  
    @Getter @Setter  
    @JsonProperty("iss")  
    private String issuerIdentifier;  
  
    @Getter @Setter  
    @JsonProperty("iat")  
    private long issuedAt;
```

```

    @Getter @Setter
    @JsonProperty("exp")
    private long expirationTime;

}

```

Dentro da classe `tokenIdClaims`, temos todas as *claims* que são retornadas após a autenticação do usuário no Identity Provider.

Para criar uma instância de `tokenIdClaims`, vamos definir um método estático que servirá como uma fábrica bem simples:

```

public static TokenIdClaims extrairClaims(
    ObjectMapper jsonMapper, OAuth2AccessToken accessToken) {
    String idToken = accessToken.getAdditionalInformation()
        .get("id_token").toString();
    Jwt tokenDecoded = JwtHelper.decode(idToken);
    try {
        return jsonMapper.readValue(tokenDecoded.getClaims(),
            TokenIdClaims.class);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

Como podemos ver no código mostrado anteriormente, não estamos validando o ID Token que estamos recebendo. Nesse caso, estamos confiando que somente tokens gerados pelo Google serão recebidos na URL de callback que definimos através do atributo `google.redirect_uri`, no arquivo `application.yml`.

Como exercício, você pode tentar realizar a validação do token conforme explicado na documentação do próprio Google:
<https://developers.google.com/identity/protocols/OpenIDConnect#validatinganidtoken>.

Voltando para a classe `OpenIdTokenServices`, adicione também o método privado `obterDateTime` para facilitar a criação de um objeto que represente a data de expiração do token quando formos salvar os dados de identificação do usuário.

```
private Date obterDatetime(long timestamp) {
    return new Date(timestamp * 1000);
}
```

Agora estamos prontos para implementar o método `saveAccessToken` da classe `OpenIdTokenServices`. Crie o método `saveAccessToken` conforme mostrado a seguir:

```
public void saveAccessToken(OAuth2AccessToken accessToken) {
    TokenIdClaims tokenIdClaims = TokenIdClaims
        .extrairClaims(jsonMapper, accessToken);

    Optional<Usuario> usuarioAutenticado = repositorioDeUsuarios
        .buscarUsuarioAutenticado(
            new IdentificadorDeAutorizacao(
                tokenIdClaims.getSubjectIdentifier())));
}
```

Através do código anterior, obtemos as *claims* do usuário e usamos o atributo `sub` para procurar em nossa base de dados um usuário que já esteja usando a conta do Google para fazer login na aplicação `bookserver`. Se o usuário já existir, apenas precisamos verificar se não há nenhum atributo que precise ser atualizado. Caso o usuário não exista, precisamos criar um novo usuário contendo os dados de autenticação.

Assim, adicione as próximas linhas no final do método `saveAccessToken` da classe `OpenIdTokenServices`:

```
Usuario usuario = usuarioAutenticado.orElseGet(() -> {
    Usuario novoUsuario = new Usuario(tokenIdClaims.getEmail(),
        tokenIdClaims.getEmail());

    new AutenticacaoOpenid(
        novoUsuario,
        new
IdentificadorDeAutorizacao(tokenIdClaims.getSubjectIdentifier()),
        tokenIdClaims.getIssuerIdentifier(),
        obterDatetime(tokenIdClaims.getExpirationTime())
    );
});
```

```
        return novoUsuario;
    });
}
```

Caso o usuário já exista, que tipo de informação podemos querer alterar? Em primeiro lugar, podemos precisar alterar a data de validade da autenticação do usuário. Portanto, adicione agora o trecho de código a seguir no método `saveAccessToken`:

```
if (usuario.getAutenticacaoOpenid().expirou()) {

    AutenticacaoOpenid autenticacaoOpenid =
    usuario.getAutenticacaoOpenid();

    autenticacaoOpenid.setValidade(obterDatetimetime(tokenIdClaims.getExpirationTime()));

}
```

Existem mais informações que podemos atribuir para um usuário autenticado? Sim, aliás é para isso que serve o serviço `userInfo`, que falei no começo do capítulo. É por meio do serviço `userInfo` que podemos obter o nome do usuário, bem como e-mail, telefone etc. No caso da aplicação `bookserver`, o atributo `nome` já basta e, para recuperar mais dados sobre o usuário autenticado, precisamos acessar o endpoint `userInfo` usando o token de acesso retornado pelo Identity Provider.

No caso da aplicação `bookserver`, é possível recuperar atributos como nome e e-mail, porque além do escopo `openid`, também estamos solicitando acesso aos escopos `profile` e `email`. Buscar os dados do usuário através do serviço `userInfo` já não é responsabilidade da classe `OpenIdTokenServices`. Para isso, vamos usar a classe `UserInfoService` injetando o atributo a seguir em `OpenIdTokenServices`.

```
@Autowired
private UserInfoService userInfoService;
```

Agora que estamos injetando um `userInfoService`, buscaremos mais informações sobre o usuário corrente. Após alterar o nome do usuário, já podemos salvar em nossa base de dados usando o objeto `repositorioDeUsuarios`, conforme mostrado a seguir.

```
Map<String, String> userInfo =  
userInfoService.getUserInfoFor(accessToken);  
String nomeDoUsuario = userInfo.get("name");  
  
usuario.alterarNome(nomeDoUsuario);  
  
repositorioDeUsuarios.registrar(usuario);
```

O método `saveAccessToken` da classe `OpenIdTokenServices` ficou um pouco grande. Caso você queira ver o código pronto usando OpenID Connect, basta importar o projeto `bookserver-openid` que está no diretório `livro-spring-oauth2/capitulo-16`.

Embora já terminamos de implementar a classe `OpenIdTokenServices`, ela conta com a colaboração de outra classe que ainda precisa ser implementada. Abra a classe `UserInfoService`, remova todo o código existente no método `getUserInfoFor` e, em seguida, adicione o seguinte trecho de código:

```
RestTemplate restTemplate = new RestTemplate();  
  
RequestEntity<MultiValueMap<String, String>> requestEntity =  
new RequestEntity<>(  
getHeader(accessToken.getValue()), HttpMethod.GET,  
URI.create(discoveryDocument.getUserInfoEndpoint()));  
  
ResponseEntity<Map> result =  
restTemplate.exchange(requestEntity, Map.class);  
  
if (result.getStatusCode().is2xxSuccessful()) {  
    return result.getBody();  
}  
  
throw new RuntimeException("Não foi possível obter os dados do usuário");
```

Não se esqueça de injetar um atributo do tipo `DiscoveryDocument` na classe `UserInfoService`, conforme mostrado a seguir.

```
@Autowired  
private DiscoveryDocument discoveryDocument;
```

16.6 Obtendo os dados de autenticação do usuário

Apesar de escrevermos bastante código para configurar tudo o que é necessário para interagir com o Google OpenID, ainda não temos nenhum ponto do código que usa as classes que criamos. Agora, precisamos efetivamente usar o serviço de autenticação do Google OpenID, e depois reconhecer o usuário da aplicação `bookserver` como autenticado.

Essa lógica pode ser implementada como um service, ou da maneira que você preferir. Entretanto, vamos criar um filtro que interceptará o acesso às áreas restritas do sistema, de modo que solicitaremos a autenticação do usuário caso este ainda não esteja logado. Abra a classe `OpenIdConnectFilter` e vamos editar o método `attemptAuthentication` que terá a lógica principal de uso do Google OpenID Connect.

Repare que a classe `OpenIdConnectFilter` estende `AbstractAuthenticationProcessingFilter`, pois assim já aproveitamos todo o mecanismo de verificação de quando o filtro deve interceptar uma requisição baseando-se em um padrão de URL.

Antes de começar a editar a lógica dessa classe, adicione as seguintes propriedades na classe `OpenIdConnectFilter`. O filtro que vamos implementar precisa da colaboração de todas as classes que estão sendo referenciadas nas propriedades a seguir.

```

@Setter
private OAuth2RestTemplate restTemplate;
@Setter
private ObjectMapper jsonMapper;
@Setter
private RepositorioDeUsuarios repositorioDeUsuarios;
@Setter
private OpenIdTokenServices tokenServices;
@Setter
private OAuth2ProtectedResourceDetails resourceDetails;

private final RequestMatcher matcherLocal;

```

Adicione o construtor do filtro conforme segue:

```

public OpenIdConnectFilter(String defaultFilterProcessesUrl, String
callback) {
    super(new OrRequestMatcher(
        new AntPathRequestMatcher(defaultFilterProcessesUrl),
        new AntPathRequestMatcher(callback)));
    this.matcherLocal = new
AntPathRequestMatcher(defaultFilterProcessesUrl);
    setAuthenticationManager(new NoopAuthenticationManager());
}

```

Agora substitua o código existente no método `attemptAuthentication` da classe `OpenIdConnectFilter` pelo código mostrado a seguir:

```

OAuth2AccessToken accessToken;

try {
    accessToken = restTemplate.getAccessToken();
    tokenServices.saveAccessToken(accessToken);

} catch (OAuth2Exception e) {
    BadCredentialsException erro = new BadCredentialsException(
        "Não foi possível obter o token", e);
    publish(new OAuth2AuthenticationFailureEvent(erro));
    throw erro;
}

```

O trecho de código anterior tenta obter um token de autenticação OpenID e, caso consiga, usa o `tokenServices` que implementamos previamente para salvar os dados do usuário autenticado no banco de dados. O interessante aqui é que, quando usamos a referência `restTemplate` para obter um token, que por sua vez é uma implementação de `OAuth2RestTemplate`, caso o token não esteja disponível no contexto atual do usuário, ele é redirecionado para a tela de autenticação do Authorization Server (que nesse caso é o Google OpenID Connect).

A lógica para o redirecionamento está definida no método `acquireAccessToken` da classe `OAuth2RestTemplate`. Caso queira entender esse fluxo, você pode analisar o próprio código do Spring Security OAuth2.

Agora pensando em um cenário em que o token de autenticação é válido, ainda precisamos extrair as *claims* do `id_token` para criar um objeto do tipo `Authentication`. É através de um objeto `authentication` que o Spring Security vai reconhecer o usuário como autenticado. Adicione o trecho de código a seguir após o código que já escrevemos para o método `attemptAuthentication`.

```
try {
    TokenIdClaims tokenIdClaims = TokenIdClaims.extrairClaims(
        jsonMapper, accessToken);

    Usuario usuario = repositorioDeUsuarios.buscarUsuarioAutenticado(
        new
    IdentificadorDeAutorizacao(tokenIdClaims.getSubjectIdentifier())).get();

    UsuarioAutenticado usuarioAutenticado = new UsuarioAutenticado(
        usuario.getAutenticacaoOpenid(), accessToken);

    Authentication authentication = new
    UsernamePasswordAuthenticationToken(
        usuarioAutenticado, null, usuarioAutenticado.getAuthorities());

    publish(new AuthenticationSuccessEvent(authentication));
    return authentication;
}
```

```

} catch (InvalidTokenException e) {
    BadCredentialsException erro = new BadCredentialsException(
        "Não foi possível obter os detalhes do token", e);
    publish(new OAuth2AuthenticationFailureEvent(erro));
    throw erro;
}

```

Para que a autenticação funcione corretamente, ainda precisamos modificar o comportamento padrão do método `doFilter` da classe `OpenIdConnectFilter`. Para isso, adicione o trecho de código a seguir, na classe `OpenIdConnectFilter`.

```

@Override
public void doFilter(ServletRequest req, ServletResponse res,
    FilterChain chain) throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;
    if (matcherLocal.matches(request)) {
        restTemplate.getAccessToken();
        chain.doFilter(req, res);
    } else {
        super.doFilter(req, res, chain);
    }
}

```

Agora já temos toda a lógica para usar o Google OpenID Connect pronta. Entretanto, ainda precisamos configurar o filtro e quando ele deverá interceptar uma requisição `http`. Portanto, abra a classe `ConfiguracaoDeSeguranca` e adicione as seguintes propriedades que deverão ser injetadas pelo Spring:

```

@.Autowired
private OAuth2RestTemplate openidRestTemplate;

@Autowired
private ObjectMapper jsonMapper;

@Autowired
private RepositorioDeUsuarios repositorioDeUsuarios;

```

```
@Autowired  
private OpenIdTokenServices tokenServices;  
  
@Autowired  
private OAuth2ProtectedResourceDetails resourceDetails;
```

Em seguida, adicione o método adiante para que o Spring possa criar um bean do tipo `OpenIdConnectFilter`.

```
@Bean  
public OpenIdConnectFilter openIdConnectFilter() {  
    OpenIdConnectFilter filter = new OpenIdConnectFilter(  
        "/livros/**", "/google/callback");  
  
    filter.setRestTemplate(openidRestTemplate);  
    filter.setJsonMapper(jsonMapper);  
    filter.setRepositoryDeUsuarios(repositoryDeUsuarios);  
    filter.setTokenServices(tokenServices);  
    filter.setResourceDetails(resourceDetails);  
    return filter;  
}
```

Agora precisamos realizar todas as configurações de segurança através do builder `HttpSecurity`. Adicione o trecho de código a seguir dentro do método `configure(HttpSecurity http)` logo após o conteúdo existente que define quais caminhos não precisam de autenticação.

```
http  
    .addFilterAfter(filtroParaClientOAuth2(),  
AbstractPreAuthenticatedProcessingFilter.class)  
    .addFilterAfter(openIdConnectFilter(),  
OAuth2ClientContextFilter.class)  
    .httpBasic()  
    .authenticationEntryPoint(new  
LoginUrlAuthenticationEntryPoint("/google/callback"))  
.and()  
    .authorizeRequests()  
    .antMatchers(caminhosPermitidos).permitAll()  
    .anyRequest().authenticated()  
.and()  
    .logout()
```

```
.logoutSuccessUrl("/")
    .permitAll()
.and()
    .csrf().disable();
```

Na configuração que estamos fazendo nesse código mostrado, estamos definindo a ordem na qual os filtros devem interceptar as requisições. Além disso, estamos definindo um *entry point* através da classe `LoginUrlAuthenticationEntryPoint`, responsável por começar um processo de login do usuário.

Após tanto código, já estamos quase prontos para testar a aplicação `bookserver`. Mas antes de mais nada, para que o trecho de código mostrado anteriormente compile, adicione o método privado a seguir.

```
private OAuth2ClientContextFilter filtroParaClienteOAuth2() {
    return new OAuth2ClientContextFilter();
}
```

O filtro `OAuth2ClientContextFilter` é responsável por iniciar o fluxo de autorização do OAuth 2.0, usando para isso o grant type que definimos quando configuramos um bean do tipo

`AuthorizationCodeResourceDetails` dentro da classe `GoogleOpenIdConnectionConfig`.

Inicie a aplicação e tente acessar a aplicação `bookserver` usando uma conta do Google. Só não se esqueça de configurar o `client_id` e o `client_secret` corretamente no arquivo `application.yml`.

16.7 Conclusão

Quando comecei a estudar o protocolo OAuth 2.0 e o Spring Security OAuth2, algumas pessoas me perguntavam como realizar autenticação usando o Google. Isso mostra claramente a confusão que existe entre usar OAuth 2.0 para uma aplicação que consome

recursos de um usuário em outra aplicação e usar OAuth 2.0 como base para um mecanismo de autenticação.

Por isso, considero este capítulo muito importante, pois além de entender as diferenças conceituais, você também tem um exemplo prático usando o Spring Security OAuth2 e o Google OpenID Connect. Muito mais poderia ser coberto sobre esse assunto. Acredito que um livro inteiro poderia ser escrito apenas sobre OpenID Connect.

A partir de agora, você já possui uma base conceitual completa para usar OAuth 2.0 e OpenID Connect em seus projetos. Entretanto, vale a pena se atentar para alguns possíveis problemas de segurança que podemos ter quando usamos OAuth 2.0. Confira o próximo capítulo que fala sobre algumas falhas de segurança conhecidas e como evitá-las.

CAPÍTULO 17

Algumas considerações sobre segurança

Ao usar um framework como o Spring Security OAuth2, toda a implementação do OAuth Provider já é fornecida restando apenas ao desenvolvedor realizar algumas configurações. Apesar da facilidade oferecida pelo framework, também é preciso se preocupar com algumas questões de segurança (por isso, é importante entender os conceitos do protocolo OAuth 2.0).

Apesar de o protocolo OAuth 2.0 já existir há bastante tempo, infelizmente muitas empresas ainda o implementam deixando algumas brechas de segurança. Muitas vezes, requisitos não funcionais como a segurança de um sistema são deixados em segundo plano, pois aparentemente o valor agregado não surte efeito tão rápido como uma nova tela bonita para o usuário. O fato de a segurança ser tão pouco priorizada acaba se refletindo até mesmo no que um programador investe em seus estudos.

Acredito que muitos que estão lendo este livro já estudaram diversos outros assuntos, como frameworks MVC, bancos de dados, bibliotecas de acesso a banco de dados, Orientação a Objetos, testes de unidade e muito mais. Mas e quanto à segurança? Quando é que nos dedicamos a estudar esse assunto?

Apesar de este livro ter o foco em como usar o protocolo OAuth 2.0 com o Spring Security OAuth2, quero alertá-lo sobre algumas possíveis armadilhas do protocolo OAuth 2.0 e como evitá-las. Além disso, quero que este capítulo seja uma inspiração para o desenvolvedor que ainda não teve a oportunidade de estudar mais sobre segurança.

Um desenvolvedor não precisa se tornar um especialista em segurança, mas entender as brechas mais comuns só aumentará a qualidade do que esse profissional vai entregar. Uma fonte

interessante para começar a estudar sobre falhas de segurança mais conhecidas é o site OWASP (*Open Web Application Security Project*).

OWASP

Essa é uma organização sem fins lucrativos focada em melhorar a segurança de software. Se trata de uma organização famosa por divulgar as *Top 10* vulnerabilidades mais críticas encontradas em sistemas Web.

Essa lista de vulnerabilidades é atualizada de tempos em tempos, sendo que durante a escrita deste livro, a última disponível era de 2013. Para conhecer as Top 10 vulnerabilidades listadas em 2013, acesse

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2013_Project.

17.1 Modelo de ameaças de segurança para o protocolo OAuth 2.0

A especificação do OAuth 2.0, conforme já citada anteriormente, não define exatamente como tudo deve ser implementado. Devido à tamanha flexibilidade, já sabemos que o título da especificação foi trocado de protocolo para framework. Conforme citado na especificação do OAuth 2.0, tal flexibilidade faz com que os aspectos de segurança que precisamos analisar dependam de muitos fatores.

O uso de um grant type em um contexto indevido, por exemplo, pode trazer problemas de segurança para os componentes envolvidos. Os componentes que estão sendo considerados aqui são o Authorization Server, Resource Server e o Client. Além disso,

as configurações sobre como usar e validar corretamente os parâmetros enviados durante a comunicação entre os componentes do OAuth 2.0 precisam ser definidas meticulosamente para evitar as brechas que vamos estudar.

Pensando em todo tipo de falha que pode ocorrer, alguns membros da comunidade que contribuíram para o desenvolvimento do protocolo OAuth 2.0 criaram um modelo de ameaças de segurança. Ele foi publicado através da RFC 6819 e pode ser acessado através do link <https://tools.ietf.org/html/rfc6819>.

MODELO DE AMEAÇAS DE SEGURANÇA

Um modelo de ameaças (*Threat Model*, em inglês) visa criar um modelo que identifica e documenta as falhas de um sistema. Um documento como este serve basicamente como um catálogo que detalha os recursos mais importantes do sistema, as probabilidades de ataque e onde estão os pontos mais vulneráveis.

Como este livro é indicado para desenvolvedores, estudaremos algumas brechas de segurança mais conhecidas e como configurar um OAuth Provider de forma segura. Também veremos um pouco sobre como um Client deve proteger credenciais e tokens de acesso. Entretanto, o maior foco será no OAuth Provider, pois é onde geralmente estão recursos valiosos do usuário.

Para se ter uma ideia, hoje em dia muitos sistemas de pagamento permitem a integração de aplicações terceiras através de OAuth 2.0. Geralmente, o uso do protocolo OAuth 2.0 nesse tipo de aplicação é uma boa escolha, pois é comum que aplicações terceiras precisem realizar operações financeiras em nome de um usuário. Imagine só a confusão que pode acontecer se um token de acesso for roubado.

No caso de um sistema de pagamentos, o dinheiro que deveria ir para uma conta X pode ser transferido para uma conta Y. Aliás, se

você estiver trabalhando em um sistema tão crítico, recomendo que você vá além do material disponível neste livro. Procure estudar a seção 10 da RFC 6749 e, se possível, a RFC 6819 que contém o modelo de ameaças do protocolo OAuth 2.0.

17.2 Análise da aplicação bookserver

Para entender os possíveis problemas de segurança que podemos encontrar ao usar o protocolo OAuth 2.0, nada melhor do que analisar na prática a própria aplicação bookserver , que viemos trabalhando até agora. Importe os projetos bookserver-all , client-code e client-implicit que estão disponíveis no diretório livro-spring-oauth2/capitulo-17 .

Essa versão da aplicação bookserver já possui configurados os grant types Authorization Code, Password, Client Credentials e Implicit. Além disso, também vamos usar as versões de Client que também estão no mesmo diretório quando formos testar os grant types Authorization Code e Implicit.

A maioria dos problemas de segurança encontrados em servidores protegidos por OAuth 2.0 pode ser evitada quando utilizamos de maneira apropriada as URLs de redirecionamento e o parâmetro state . Abra a classe ConfiguracaoOAuth2.java e perceba que em nenhum momento registramos uma URI de redirecionamento para os Clients que configuramos.

```
clients.inMemory()
    .withClient("cliente-app")
    .secret("123456")
    .scopes("read", "write")
    .resourceIds(RESOURCE_ID)
    .authorizedGrantTypes(
        "password", "authorization_code")
    .and()
```

```
.withClient("cliente-admin")
.secret("123abc")
.authorizedGrantTypes("client_credentials")
.scopes("read")
.resourceIds(RESOURCE_ID)
.and()
.withClient("cliente-browser")
.authorizedGrantTypes("implicit")
.scopes("read");
}
```

Quanto ao parâmetro `state`, o Spring Security OAuth2 já fornece suporte para retornar o valor desse parâmetro concatenado na URL de redirecionamento, caso o Client informe-o na etapa de autorização. Quando usamos o Spring Security OAuth2, o uso do parâmetro `state` fica a cargo do Client. Entretanto, tenha em mente que, sempre que estiver implementando um servidor OAuth 2.0, é necessário que o servidor dê suporte ao parâmetro `state` para evitar ataques CSRF.

17.3 A importância da URI de redirecionamento

Inicialmente, vamos estudar como a falta de registro de uma URL de redirecionamento pode ser perigosa, principalmente quando se trata do grant type Implicit, em que o Client não se autentica no Authorization Server. Apenas para relembrar, quando usamos o grant type Implicit, o Authorization Server nem ao menos deve gerar uma credencial para o Client, pois esse grant type é indicado para aplicações públicas, ou seja, aplicações que não têm condições de manter a confidencialidade das credenciais. Imagine o quanto desastroso pode ser deixar exposto no browser o `client_secret` de um Client.

Então, como o Authorization Server pode ter a garantia de que está entregando um token de acesso para o Client correto? Como um

Client público não se autentica no Authorization Server, não é possível saber quem está de fato solicitando um token de acesso. O máximo que se pode fazer é devolver o token, usando a URL de redirecionamento cadastrada pelo Client durante o registro no Authorization Server.

Para ajudar você a entender por que a URI de redirecionamento é tão importante, vamos iniciar as aplicações `bookserver-all` e `client-implicit` e simular um roubo de token de acesso, conforme mostrado na figura a seguir.

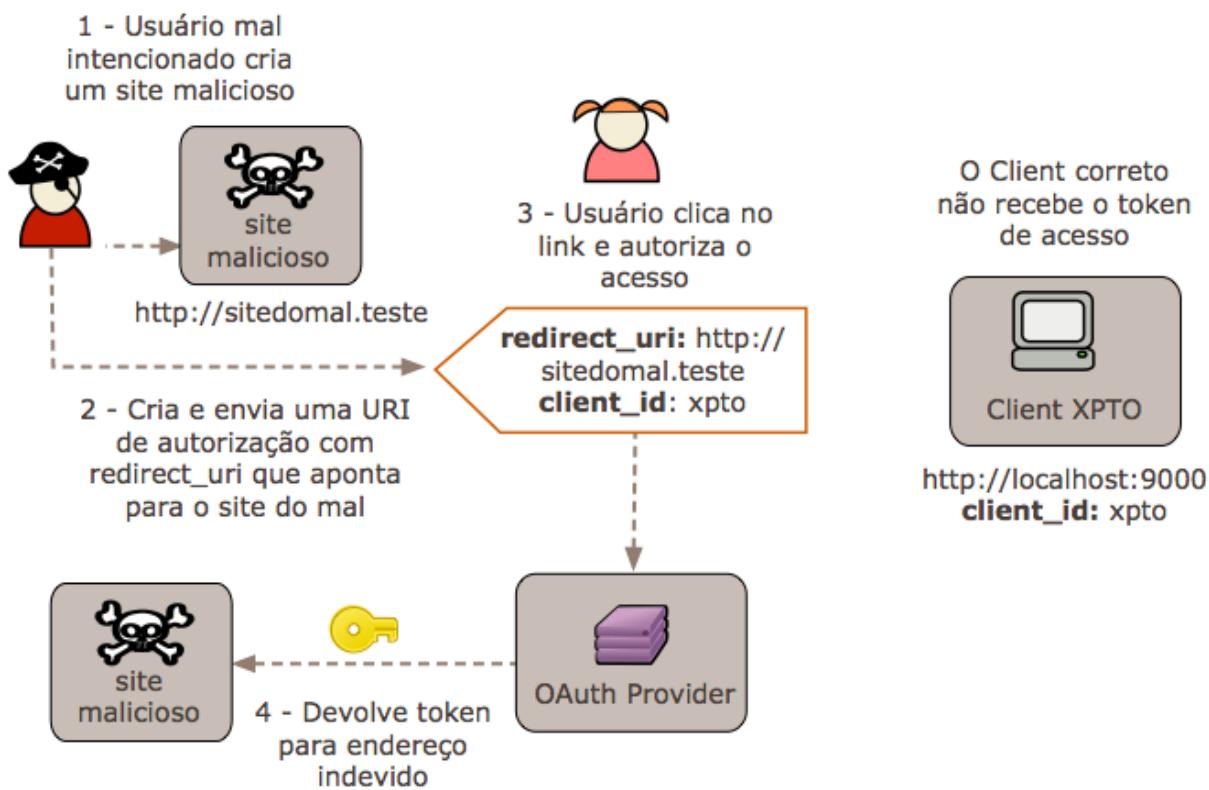


Figura 17.1: Roubo de token de acesso trocando o parâmetro `redirect_uri`

No cenário de ataque que vamos estudar agora, o usuário é induzido a clicar em um link que nada mais é do que uma URL de autorização forjada por um usuário mal-intencionado. Como essa URL contém o `client_id` de uma aplicação que o usuário teoricamente confia, mesmo que o usuário não esteja logado nessa aplicação, ele ainda assim pode autorizar o uso de seus dados.

Diante dessa situação, um token de acesso pode facilmente ser entregue para um site qualquer, ou até mesmo um Client que está sob o poder de um usuário mal-intencionado.

Uma vez que os projetos `bookserver-all` e `client-implicit` tenham sido inicializados, vamos primeiramente criar uma conta de usuário que vai representar o usuário mal-intencionado na aplicação `client-implicit`, conforme mostrado na figura seguinte.

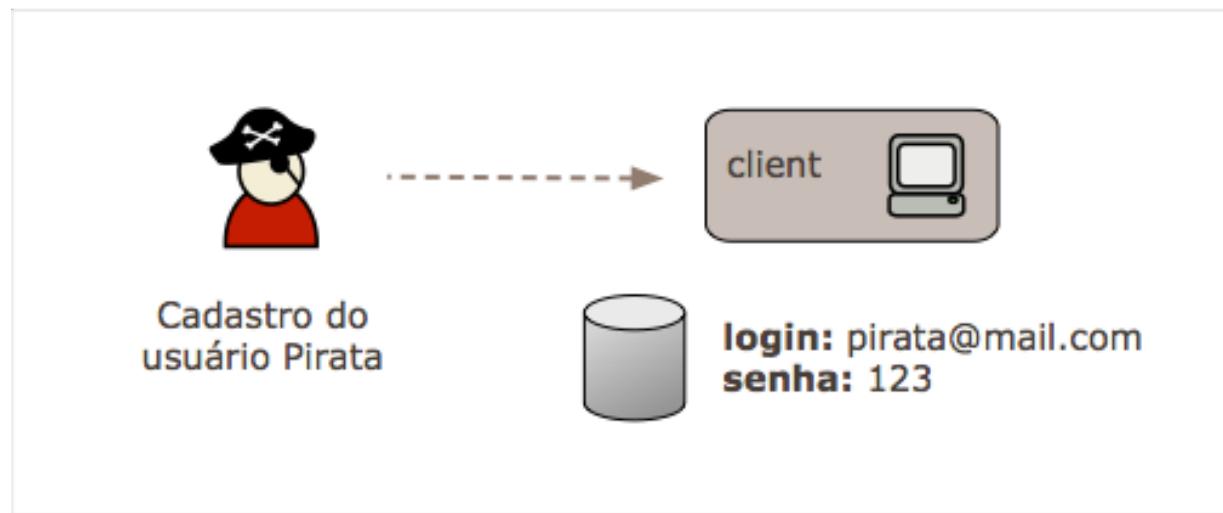


Figura 17.2: Cadastro do usuário mal-intencionado

Na imagem anterior, mostro a criação de um usuário que batizei de `Pirata` , com o login `pirata@mail.com` e a senha `123` .

Agora para forjarmos uma URL de autorização que permita o redirecionamento do usuário (vítima) com o token de acesso para o site ou Client que está sob o poder do `Pirata` , vamos utilizar o navegador Firefox com o plugin **NoRedirect**, que pode ser visto na imagem a seguir. Para gerenciar os plugins do Firefox, basta digitar `about:addons` na barra de endereços do navegador.

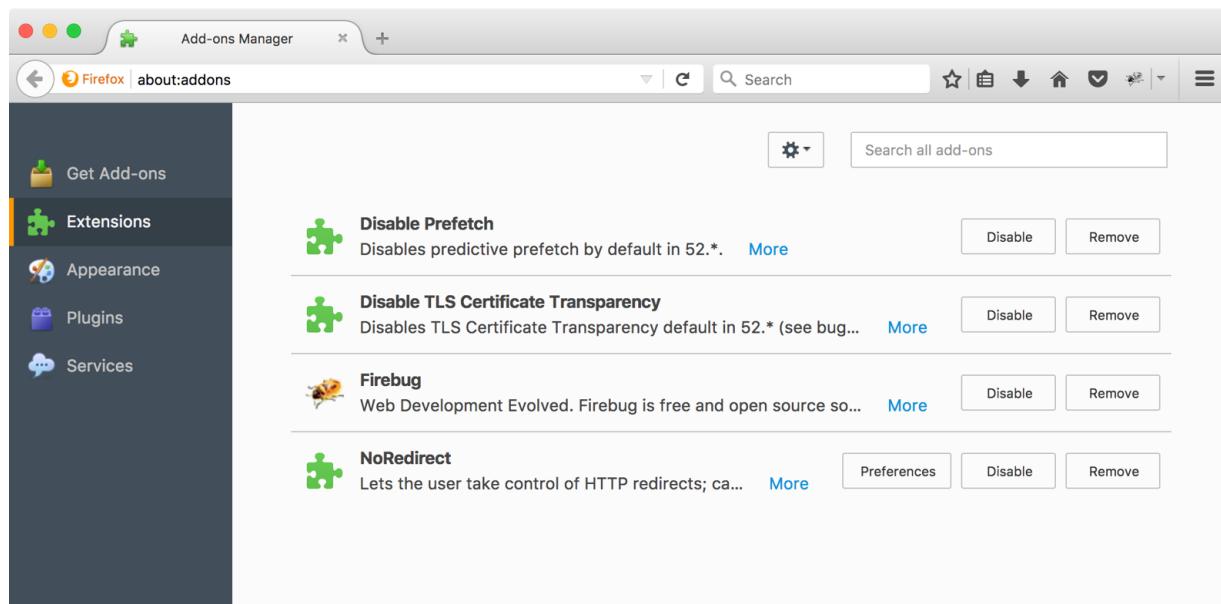


Figura 17.3: Navegador Firefox com o plugin NoRedirect

Com esse plugin, os redirecionamentos não serão feitos automaticamente, nos permitindo interceptar qualquer redirecionamento para modificar o que for necessário na URL. Para que o plugin funcione em nosso cenário de teste, precisamos adicionar algumas regras através do plugin `NoRedirect`. Para isso, clique no botão `Preferences` do plugin para acessar a seguinte tela:

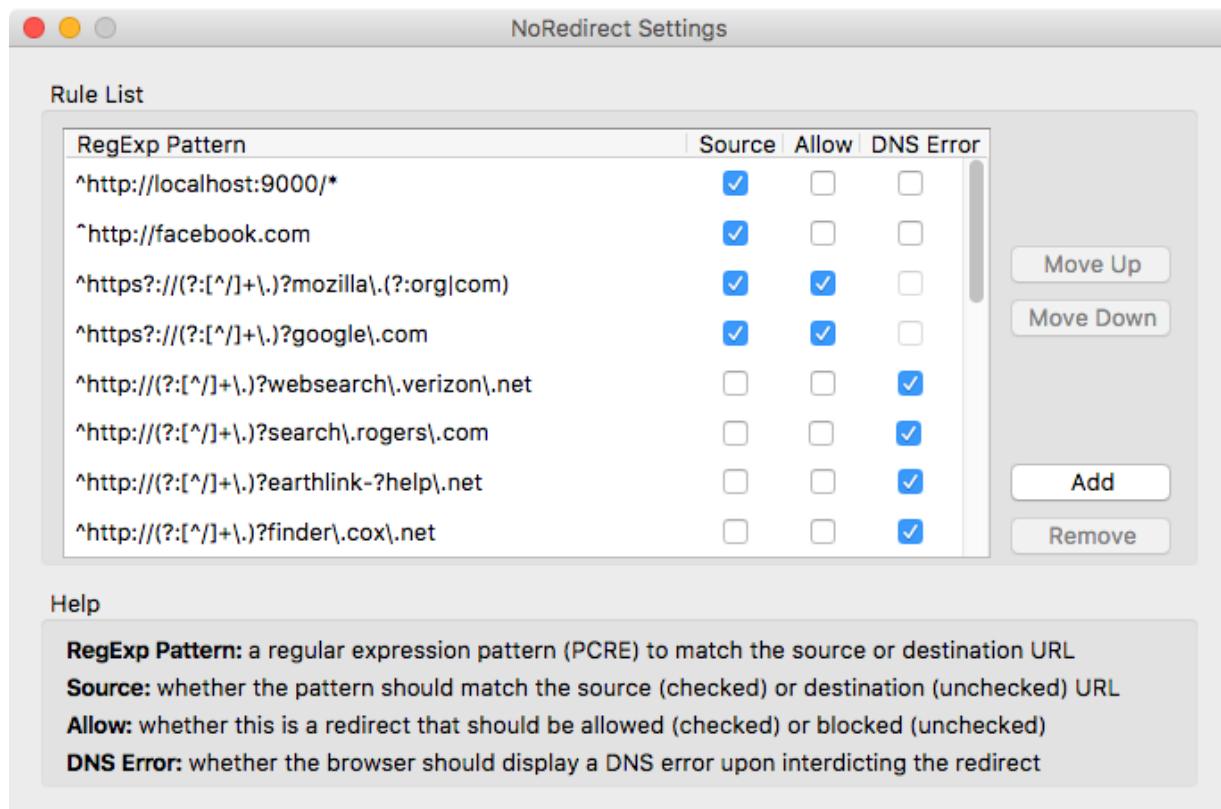


Figura 17.4: Tela de configuração do NoRedirect

Vamos adicionar uma regra para interceptar qualquer redirecionamento para a aplicação `client-implicit`. Para isso, clique no botão `Add` e digite o seguinte padrão de URL:

`^http://localhost:9000/integracao/implicit`

Em seguida, marque a opção `Source` e mova essa regra para o começo da lista de regras.

Agora vamos acessar a URL <http://localhost:9000> e fazer o login com o usuário `Pirata` que acabamos de cadastrar. Perceba que você não será redirecionado automaticamente para a tela inicial da aplicação `client-implicit`. Em vez disso, aparecerá um link na tela, conforme mostrado na figura a seguir.

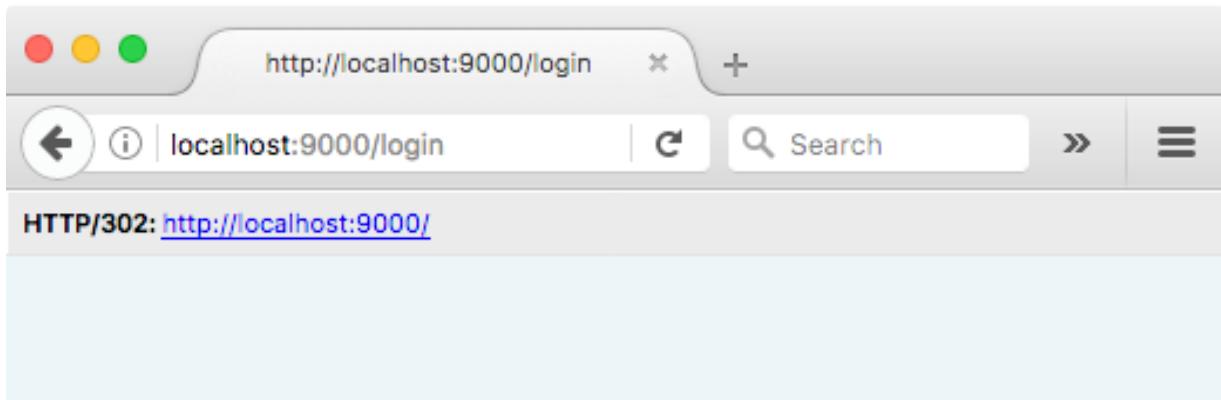


Figura 17.5: Tela com o link de redirecionamento

Clique no link `http://localhost:9000/` que aparece na tela e, em seguida, clique no botão `Minha Conta` para acessar a tela que mostra as atividades do usuário na aplicação Client. Para começar a interagir com a aplicação `bookserver` e capturar a URL de autorização que vamos alterar, clique no link mostrado na figura seguinte, destacado em azul.

! Você tem conta no bookserver?

Mostre [aqui](#) os livros que você já leu para melhorar seu perfil.

Figura 17.6: Acesso via OAuth 2.0

Mais uma vez, em vez de o navegador redirecionar o usuário automaticamente para a aplicação `bookserver`, o plugin `NoRedirect` gera um link que dá a opção de fazer ou não o redirecionamento. Em vez de clicar no link como fizemos no login, copie-o para analisarmos o seu conteúdo que deve ser algo parecido com o mostrado a seguir:

```
http://localhost:8080/oauth/authorize?  
scope=read&response_type=token&redirect_uri=http%3A%2F%2Flocalhost%3A9000%  
2Fintegracao%2Fimplicit&client_id=cliente-browser
```

Olhando para a URL anterior, que se trata do endpoint de autorização da aplicação `bookserver`, um usuário mal-intencionado

tem tudo o que precisa para obter um token de acesso de algum usuário que já esteja logado na aplicação `bookserver`. Imagine que esse usuário mal-intencionado criou uma aplicação web cuja URL é `http://sitedomal.teste`, e que essa aplicação seja capaz de extrair um token de acesso que chegue através de um fragmento de URL.

Tudo que esse usuário mal-intencionado precisa fazer é alterar o parâmetro `redirect_uri` para algo como

`http://sitedomal.teste/callback`, criando uma nova URL que aponta para o endpoint de autorização da aplicação `bookserver`.

```
http://localhost:8080/oauth/authorize?  
scope=read&response_type=token&redirect_uri=http%3A%2F%2Fsitedomal.teste%2  
Fcallback&client_id=cliente-browser
```

O usuário mal-intencionado pode enviar essa URL para várias pessoas que tenham conta na aplicação `bookserver`. Se uma delas clicar nesse link e estiver logada na aplicação `bookserver`, ela será redirecionada para a aplicação maliciosa com o token de acesso concatenado na URL, como um fragmento. Resumindo: o token será entregue para um Client criado por um usuário mal-intencionado que agora pode realizar operações em nome do usuário que clicou no link malicioso.

Vamos fazer o teste acessando a URL anterior, mas antes de tudo, faça o login na aplicação `bookserver` com qualquer usuário que não seja o `Pirata`. Com um usuário logado em `bookserver`, o navegador vai lhe redirecionar para a tela de autorização. Ao autorizar o acesso à aplicação Client, o usuário (vítima) será redirecionado para o site `http://sitedomal.teste` com o token de acesso disponível como um fragmento na URL.

```
http://sitedomal.teste/callback/implicit#access_token=779fdcf5-7a57-4af1-  
8f30-047dce981863&token_type=bearer&expires_in=40859
```

É bem provável que o usuário autorize o acesso ao Client caso este se trate de uma aplicação que a vítima conhece e confia.

Para evitar que um usuário altere a URL de autorização como fizemos, basta exigir que a aplicação Client cadastre uma URI de redirecionamento durante o registro. Vamos editar a classe `ConfiguracaoOAuth2` do projeto `bookserver-all` para cadastrar uma URI de redirecionamento para a aplicação `client`, cujo `client_id` definimos como `cliente-browser`. A configuração deve ficar da seguinte forma:

```
.withClient("cliente-browser")
.authorizedGrantTypes("implicit")
.redirectUris("http://localhost:9000/integracao/implicit")
.scopes("read");
```

Vamos iniciar as aplicações novamente e tentar realizar o mesmo tipo de ataque alterando o parâmetro `redirect_uri` para o *site do mal*. Ao tentar acessar a URL maliciosa, você receberá a seguinte mensagem de erro na aplicação `bookserver`:

```
error="invalid_grant"
error_description="Invalid redirect: http://sitedomal.teste/callback does
not match one of the registered values:
[http://localhost:9000/integracao/implicit]"
```

É interessante observar que o erro foi apresentado diretamente na aplicação `bookserver`, ou seja, o usuário não foi redirecionado para a URL que foi adicionada no parâmetro `redirect_uri`. Isso evita que o usuário seja redirecionado para uma aplicação mal-intencionada. Bastou registrar uma URL de redirecionamento para o Client, para melhor proteger os tokens de acesso de serem roubados.

Considerando o grant type Implicit, como o token fica exposto na URL de redirecionamento, é importante que o Client solicite tokens com o menor escopo possível e que o Authorization Server gere tokens com um tempo de validade curto o bastante para minimizar os impactos de uma fraude caso um seja roubado.

Validação completa do parâmetro `redirect_uri`

Em se tratando da URI de redirecionamento, além de obrigar o seu uso, também é importante que a validação por parte do servidor faça um *match* completo do parâmetro `redirect_uri` com o que foi cadastrado no Authorization Server. Utilizar validação parcial (considerando apenas o domínio por exemplo) pode permitir que um usuário mal-intencionado altere a URI de redirecionamento para uma página criada com o intuito de roubar tokens de outros usuários.

Nesse caso, o usuário mal-intencionado poderia criar uma página dentro do próprio Client, seja através de injeção de código ou mesmo uma funcionalidade do próprio site. Uma plataforma de blog, por exemplo, poderia facilmente ter esse tipo de vulnerabilidade caso uma URI de redirecionamento seja validada parcialmente.

17.4 A importância do parâmetro state

Ainda que façamos o uso correto do parâmetro `redirect_uri`, um usuário mal-intencionado pode utilizar a técnica de CSRF (*Cross Site Request Forgery*) para fazer com que um usuário comum passe a usar o token do usuário mal-intencionado. Pensando na integração entre as aplicações `bookserver` e `client`, imagine que quanto mais livros um usuário puder exibir no Client, melhor a reputação desse usuário. Sabendo disso, um usuário mal-intencionado poderia substituir o token de um usuário X pelo seu próprio token e não cadastrar livro algum. Dessa forma, todos os livros que o usuário X cadastrasse nunca seriam exibidos na aplicação Client, tendo sua reputação prejudicada.

Mas como esse tipo de ataque pode ser realizado? Vamos analisar a figura a seguir que apresenta um fluxo completo de como um Resource Owner X pode substituir o token de um Resource Owner Y.

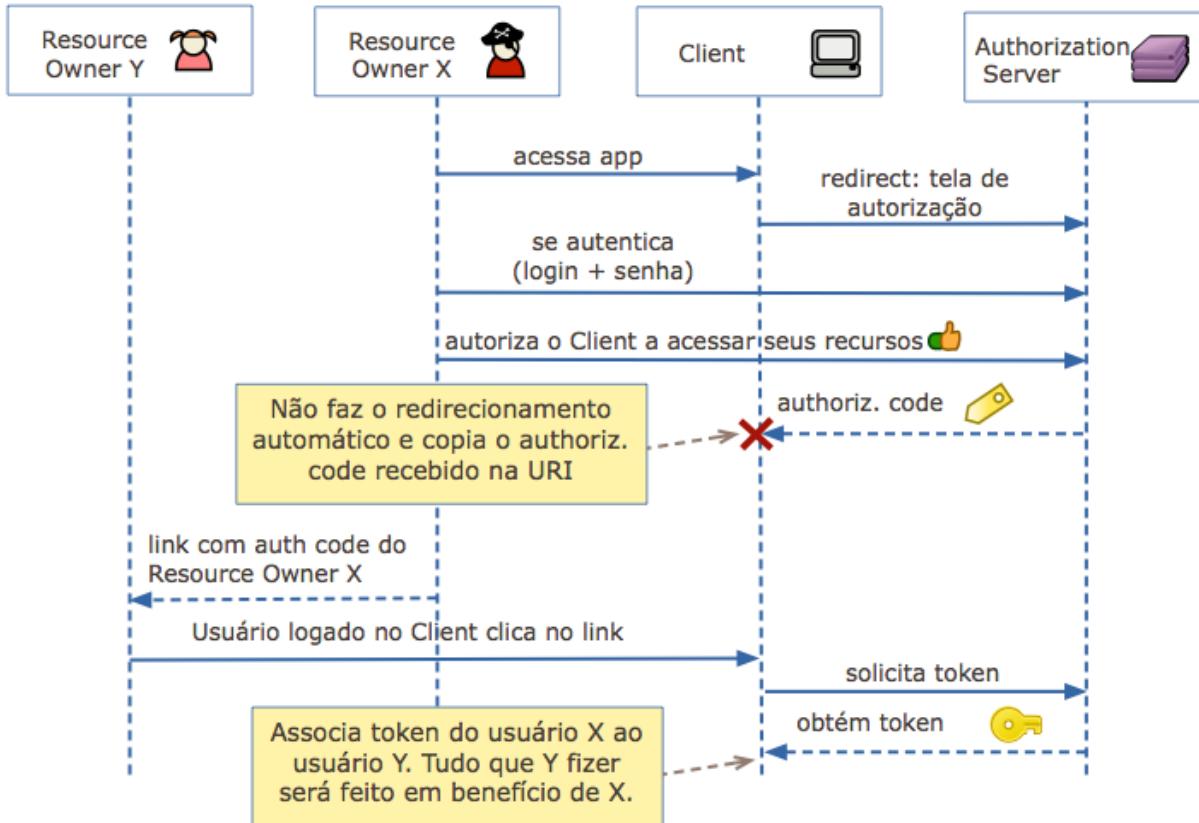


Figura 17.7: Ataque CSRF para alterar o token de um usuário

Para seguir esse fluxo de ataque, vamos trabalhar com o projeto `client-code` que também está dentro do diretório `livro-spring-oauth2/capitulo-17`. Essa implementação de Client utiliza o grant type Authorization Code.

Inicie as aplicações `bookserver-all` e `client-code` e, antes de começarmos os testes, vamos primeiro criar duas contas de usuário em cada aplicação. No meu caso, criei os seguintes usuários na aplicação `bookserver-all`:

- Princesa Jujuba com o login `jujuba@mailinator.com` e senha `123`;
- Pirata com o login `pirata@mailinator.com` e senha `123`.

Na aplicação `client-code`, criei os seguintes usuários. Se você já cadastrou o usuário Pirata na aplicação `client-implicit`, não será

preciso cadastrar novamente agora, pois estamos usando o mesmo banco de dados.

- Princesa Jujuba com o login `jujuba@mail.com` e senha `123`;
- Pirata com login `pirata@mail.com` e senha `123`.

A seguir, vamos alterar a regra de interceptação de redirecionamento que fizemos anteriormente através do plugin `NoRedirect`, substituindo a expressão `^http://localhost:9000/*` para `^http://localhost:8080/*`.

Agora acesse a URL <http://localhost:9000> a partir do Firefox e vamos fazer o login com o usuário `pirata@mail.com`. Assim como fizemos no teste anterior, vamos clicar no link que realiza a integração via OAuth 2.0, conforme mostrado na figura a seguir.

! Você tem conta no bookserver?
Mostre [aqui](#) os livros que você já leu para melhorar seu perfil.

Figura 17.8: Acesso via OAuth 2.0

Ao ser redirecionado para o endpoint de autorização da aplicação `bookserver`, vamos nos logar com o usuário `pirata@mailinator.com` e fornecer as devidas permissões. Como estamos usando o plugin `NoRedirect`, quando o Authorization Server nos redirecionar de volta para o Client, podemos obter o parâmetro `code` presente na URL de redirecionamento. Veja:

<http://localhost:9000/integracao/callback?code=tFsP1Y>

Copie a URL que apareceu para você e feche o browser para encerrar todas as sessões abertas no momento. Agora para simular um ataque CSRF, vamos nos logar com o usuário `jujuba@mail.com` na aplicação Client (<http://localhost:9000>) e acessar a URL de callback que contém o parâmetro `code`, obtido durante o fluxo de autorização realizado pelo usuário `Pirata`.

Ao fazer isso, todos livros que o usuário `Pirata` cadastrar serão exibidos na conta do usuário `jujuba`. Nesse caso, o usuário mal-intencionado pode causar uma verdadeira bagunça no sistema.

Como podemos evitar esse tipo de problema? O parâmetro `state` serve justamente para isso. Nesse caso, tudo que precisamos fazer é alterar o código do `client-code` para enviar o parâmetro `state` na URL de autorização, e depois verificar se o `state` retornado na URL de callback é igual ao valor enviado na etapa de autorização.

Antes de mais nada, vamos criar uma classe que manterá o atributo `state` enquanto o usuário é redirecionado do Client para o Authorization Server. Crie a classe `OAuth2State` no pacote `br.com.casadocodigo.integracao.bookserver` do projeto `client-code`.

```
@Component
@Scope(
    value = WebApplicationContext.SCOPE_SESSION,
    proxyMode = ScopedProxyMode.TARGET_CLASS)
public class OAuth2State {

    @Getter @Setter
    private String state;

    public boolean temStateValido(String state) {
        if (this.state != null) {
            return this.state.equals(state);
        }
        return true;
    }
}
```

Em seguida, vamos editar o método `getAuthorizationEndpoint` da classe `AuthorizationCodeTokenService` para adicionar o parâmetro `state` na URL de autorização.

```
UUID identificadorRandomico = UUID.randomUUID();
String state = new String(Base64.getEncoder().encode(
    identificadorRandomico.toString().getBytes()));
```

```
oAuth2State.setState(state);
parametros.put("state", oAuth2State.getState());
```

Não se esqueça de injetar o componente `oAuth2State` que acabamos de criar na classe `AuthorizationCodeTokenService`.

```
@Autowired
private OAuth2State oAuth2State;
```

Agora que estamos enviando o parâmetro `state`, o Authorization Server devolverá o mesmo valor na URL de redirecionamento. Só precisamos validar se o `state` recebido na URL de callback é igual ao `state` gerado na fase de autorização. Abra a classe `IntegracaoController` e injete o componente `oAuth2State`, assim como fizemos para a classe `AuthorizationCodeTokenService`.

Depois, adicione a seguinte validação no método `callback` da classe `IntegracaoController`:

```
@RequestMapping(value = "/callback", method = RequestMethod.GET)
public ModelAndView callback(String code, String state) {

    if (!oAuth2State.temStateValido(state)) {
        throw new RuntimeException("Parametro state inválido");
    }

    // resto do código omitido
}
```

Apesar de termos uma validação rudimentar que não apresenta uma mensagem adequada para o usuário, agora o tipo de ataque por CSRF já não pode mais ser executado.

Com poucas configurações, já estamos protegendo tanto o Authorization Server quanto o Client dos principais ataques em sistemas que usam o protocolo OAuth 2.0. Apesar de serem configurações simples, acredite que existem sistemas que ainda não realizam as validações que estudamos até agora.

17.5 Detalhes importantes para o Client

Além das vulnerabilidades que estudamos até agora, também é preciso se preocupar como credenciais e tokens são protegidos pelo Client. De nada adianta usar TLS/SSL para proteger um token sendo trafegado durante uma requisição se este é armazenado sem nenhum mecanismo de criptografia na base de dados do Client. Podemos dizer o mesmo das credenciais do Client que não devem ser expostas na aplicação. Por isso apenas Clients do tipo confidencial devem receber `client_secret` durante o registro no Authorization Server.

Outra maneira de expor tokens de acesso indevidamente é através de requisições que não usam o header `Authorization`. De acordo com a especificação do protocolo OAuth 2.0, o Resource Server pode receber os tokens de acesso como um parâmetro de URL ou como um campo de formulário. A pior abordagem para acessar um recurso protegido por OAuth 2.0 é enviando o token através de parâmetro de URL. O token pode ser facilmente gravado em arquivos de log do servidor (arquivo `access log`) bem como pode ser cacheado em um Proxy.

Pensando em aplicações nativas, uma dica interessante para evitar que credenciais sejam capturadas é utilizar o registro dinâmico do Client, de forma que uma aplicação mobile não precise manter em seu código-fonte o `client_id` e o `client_secret`, por exemplo. A RFC 7591 define um profile OAuth 2.0 para implementação de um OAuth Provider que permite o registro dinâmico de Clients. Esta RFC pode ser acessada através do link <https://tools.ietf.org/html/rfc7591>.

17.6 Conclusão

Chegamos ao final do livro com algumas dicas de segurança que acredito que não podem faltar quando se fala em OAuth 2.0. Diante

de prazos apertados e o grande número de tecnologias que precisamos estudar, realmente fica difícil estudar a fundo um assunto como o protocolo OAuth 2.0.

Com tantas especificações complementares espalhadas, nas quais muitas vezes é difícil encontrar material que compile teoria e prática, é tentador ficar na superficialidade, contentando-se apenas com códigos de exemplos encontrados pela internet que parecem resolver os problemas rapidamente. Ao estudar este livro e este pequeno capítulo sobre segurança, tenho certeza de que você olhará para sua aplicação com outros olhos, e perceberá a importância de adicionar o mínimo de segurança nos projetos em que estiver trabalhando.

Espero que este material tenha ajudado você a utilizar o framework OAuth 2.0 em seu projeto de forma confiante, pois agora você tem conhecimento teórico e prático aprofundado sobre esse protocolo tão querido por alguns e odiado por outros.

CAPÍTULO 18

Bibliografia

BROOK, Chris. *Paypal fixes OAuth token leaking vulnerability*. Nov. 2016. Disponível em: <https://threatpost.com/paypal-fixes-oauth-token-leaking-vulnerability/122136/>.

EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

HAMMER, Eran. *OAuth 2.0 and the Road to Hell*. Jul. 2012. Disponível em: <https://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell>.

MICROSOFT CORPORATION. *Modelagem de ameaças de segurança*. Jan. 2014. Disponível em: <https://technet.microsoft.com/pt-br/library/dd569893.aspx>. Acesso em: Maio de 2017.

NEWMAN, Sam. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

RICHER, Justin; SANSO, Antonio. *OAuth 2 In Action*. Manning Publications, 2017.

RICHER, Justin. *User Authentication with OAuth 2.0*. Disponível em: <https://oauth.net/articles/authentication/>. Acesso em: Maio de 2017.

SAKIMURA, N.; BRADLEY, J.; JONES, M.; JAY, E. *OpenID Connect Discovery 1.0 incorporating errata set 1*. Nov. 2014. Disponível em: https://openid.net/specs/openid-connect-discovery-1_0.html. Acesso em: Maio de 2017.

SAKIMURA, N.; BRADLEY, J.; JONES, M.; MEDEIROS, B.; MORTIMORE, C. *OpenID Connect Core 1.0 incorporating errata set 1*. Nov. 2014. Disponível em: http://openid.net/specs/openid-connect-core-1_0.html. Acesso em: Maio de 2017.

SAKIMURA, N.; BRADLEY, J.; JONES, M.; MEDEIROS, B.; MORTIMORE, C. *OpenID Connect Basic Client Implementer's Guide 1.0 - draft* 37. Ago. 2015. Disponível em: http://openid.net/specs/openid-connect-basic-1_0.html. Acesso em: Maio de 2017.

SIRIWARDENA, Prabath. *JWT, JWS and JWE for Not So Dummies!* Abr. 2016. Disponível em: <https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3>. Acesso em: Maio de 2017.

SIRIWARDENA, Prabath. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Appress, 2017.

PARASCHIV, Eugen. *Using JWT with Spring Security OAuth*. Jul. 2017. Disponível em: <http://www.baeldung.com/spring-security-oauth-jwt>. Acesso em: Maio de 2017.

VERNON, Vaughn. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

Links

Google Identity Platform, sobre OpenID Connect –
<https://developers.google.com/identity/protocols/OpenIDConnect>

MDN web docs, sobre HTTP access control (CORS) –
https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Spring Security OAuth2, sobre OAuth 2 Developers Guide –
<http://projects.spring.io/spring-security-oauth/docs/oauth2.html>